13. Temporal struct members

13.1 Events

The ***e*** language provides temporal constructs for specifying and verifying behavior over time. All ***e*** temporal language features depend on the occurrence of events, which are used to synchronize activity with a simulator and within the ***e*** program.

13.1.1 Causes of events

Table 26 describes how an event is made to occur.

Table 26—Event causation

Syntax	Cause of the event
event a **is** (**@**b and **@**c)**@**d	Derived from other events (see Clause 12).
event a **is rise**('top.b')**@sim**	Derived from behavior of a simulated device (see Clause 12).
event a**;** meth_b()**@**c **is** { ... ; **emit** a; ... };	By the **emit** action in procedural code (see 13.1.3.2).

13.1.2 Scope of events

Events are defined as a part of a struct definition. When a struct is instantiated, each instance has its own event instances. Each event instance has its own schedule of occurrences. There is no relation between occurrences of event instances of the same type. All references to events are to event instances.

The scoping rules for events are similar to other struct members, such as *fields*.

— If a path is provided, use the event defined in the struct instance pointed to by the path.
— If no path is provided, the event is resolved at compile time. The current struct instance is searched.
— If the event instance is not found, a compile-time error shall be issued.

13.1.3 Defining and emitting named events

This subclause describes the **event** and **emit** constructs.

13.1.3.1 event

Purpose	Define a named event		
Category	Struct member		
Syntax	**event** *event-type*[**is** *temporal-expression* [**using** *temporal-operators*]] **event** *event-type*[**is only** *temporal-expression*] [**using also** *temporal-operators*]		
Parameters	*event-type*	The name of the event type (any legal ***e*** identifier).	
	temporal-expression	An event or combination of events and temporal operators. See also Clause 12.	
	temporal-operators	One or more temporal operators of the form: *operation condition*, separated by commas, where:	
		operation is one of the following keywords:	
		abort	Terminate evaluation of the temporal expression in the current tick and restart it in the next tick.
		start	Start the evaluation of the temporal expression in the current tick if the event is currently in a stopped state.
		exclusive_start	Similar to start, but in addition the event is stopped when the struct is created, and remains stopped until acted upon by the exclusive start.
		stop	Stop the evaluation of the temporal expression in the current tick. It will remain stopped until a start instruction is issued.
		condition is one of the following:	
		@event-name	Triggering temporal event. This event can belong to the context struct (that is, [**me**.]event-name), or reachable via a constant path (that is, *const-or-unit-instance-field-path.event-name*).
		none	Return to the default state. This value is intended for use when there was a previous triggering event for this instruc-tion that should be removed.
		empty	Used to specify a dummy event (i.e., an event that never happens), for use with the procedural API - see (Yuri/AlanH: Add XREF to new section once done).

Events can be attached to TEs, using the **is** *temporal-expression* syntax, or they can be unattached. An *attached event* is emitted automatically during any tick in which the TE attached to it succeeds. For a definition of the success of a TE, see 12.3. If an event has been attached to a TE, one or more temporal operators can also be attributed to it, with the **using** *temporal-operators* syntax

Events, like methods, can be redefined in struct extensions. The **is only** *temporal-expression* syntax is used to change the definition of an event, e.g., to define an event once and then attach it to several different TEs under different **when** struct subtypes. The **using also** *temporal-operators* syntax is used to add a temporal operator in a redefinition of the event (when used together with **is only**), or to add it as a separate extension to an existing event (when used without **is only**).

13.1.3.2 emit

Purpose	Cause a named event to occur	
Category	Action	
Syntax	**emit** [*struct-exp*.]*event-type*	
Parameters	*struct-exp*	An expression referring to the struct instance in which the event is defined.
	event-type	The type of event to emit.

This causes an event of the specified type to occur.

— Emitting an event causes the immediate evaluation of all TEs containing that event.

— The **emit** event does not consume time. It can be used in regular methods and in TCMs.

— The simplest usage of **emit** is to synchronize two TCMs, where one TCM waits for the named event and the other TCM emits it.

Syntax example:

```
emit ready
```

13.1.4 Predefined events

Predefined events are emitted at particular points in time.

13.1.4.1 General predefined events

Table 27 lists the general predefined events.

Table 27—Predefined events

Predefined event	Description
sys.any	Emitted on every tick.
sys.tick_start	Emitted at the start of every tick.
sys.tick_end	Emitted at the end of every tick.
session.start_of_test	Emitted once at test start.
session.end_of_test	Emitted once at test end.
struct.quit	Emitted when a struct's **quit()** method is called. Only exists in structs that contain events or have members that consume time (for example, TCMs or **on** struct members).
sys.new_time	In stand-alone operation (no simulator), this event is emitted on every **sys.any** event. When a simulator is being used, this event is emitted whenever a callback occurs and the attached simulator's time has changed since the previous callback.

13.1.4.1.1 sys.any

This event is a special event that defines the finest granularity of time. The occurrence of any event in the system causes an occurrence of the **any** event at the same tick. In stand-alone ***e*** program operation (that is, with no simulator attached), the **sys.any** event is the only one that occurs automatically. It typically is used as the clock for stand-alone operation.

13.1.4.1.2 sys.tick_start

This event is provided mainly for visualizing and debugging the program flow.

13.1.4.1.3 sys.tick_end

This event is provided mainly for visualizing and debugging the program flow. It also can be used to provide visibility into changes of values that are computed during the tick, such as the values of coverage items.

13.1.4.1.4 session.start_of_test

The first action the predefined **run()** method executes is to emit the **session.start_of_test** event. This event is typically used to anchor TEs to the beginning of a test.

13.1.4.1.5 session.end_of_test

This event is typically used to sample data at the end of the test. This event cannot be used in TEs, as it is emitted after evaluation of TE has been stopped. The **on session.end_of_test** struct member is typically used to prepare the data sampled at the end of the test.

13.1.4.1.6 struct.quit

This only exists in structs that contain temporal members (events, **on**, **expect**, or TCMs). It is emitted when the struct's **quit()** method is called, to signal the end of time for the struct.

The first action executed during the check test phase is to emit the **quit** event for each struct that contains it. This event can be used to cause the evaluation of TEs that contain the **eventually** temporal operator (and check for **eventually** TEs that have not been satisfied).

13.1.4.1.7 sys.new_time

This event is emitted on every **sys.any** event in stand-alone operation (no simulator). When a simulator is being used, this event is emitted whenever a callback occurs and the attached simulator's time has changed since the previous callback.

13.1.4.2 Simulation time and ticks

Using any of the following expressions causes the DUT to be monitored for a change in that expression:

— **rise** | **fall** | **change** (*HDL expression*) **@sim**
— **wait delay** *expression*
— Verilog event

For each simulation delta cycle where a change in at least one of these monitored expressions occurs, the simulator passes control to the ***e*** program. If simulation time has advanced since the last time control was passed to the ***e*** program, a **new_time** event is issued. In any case, **tick_start** and **any** events are issued.

Then, after emitting all events initiated by changes in monitored expressions in that simulation delta cycle, a **tick_end** event is issued.

Thus, the **new_time** event corresponds to a new simulation time slot and a tick corresponds to a simulation delta cycle where at least one monitored expression changes.

Multiple ticks can occur in the same simulation time slot under the following conditions:

— When a new value is driven into the DUT and that value causes a change in a monitored HDL object, as in a clock generator

— When a monitored event is derived from another monitored event, as in a clock tree

— When a zero-delay HDL subprogram is called from ***e***

For an explanation of when values are assigned, see 19.5.

NOTE—Glitches that occur in a single simulation time slot are ignored; only the first occurrence of a particular monitored event in a single simulation time slot is recognized.

13.2 on

Purpose	Specify a block of actions that execute on an event	
Category	Struct member	
Syntax	**on** [*const-exp.*]*event-type* **{***action***; ...}**	
Parameters	*const-exp*	An optional expression identifying the struct or unit in which event-type is defined. This expression must remain constant and thus can consist only of: — A **const** field — A unit instance name — Indexing a unit from a list of unit instances (a constant unit pointer) If not provided, the context struct or unit is assumed.
	event-type	The name of an event that invokes the action block.
	*action***; ...**	A block of non-time-consuming actions.

This defines a struct member that executes a block of actions immediately whenever a specified event occurs. An **on** struct member is similar to a regular method, except that it is invoked automatically upon an occurrence of the event. An **on** action block is executed before TCMs waiting for the same event. The actions are executed in the order in which they appear in the action block.

To extend an **on** struct member, its declaration can be repeated and a different action block used. This has the same effect as using **is also** to extend a method.

If no *const-exp* is specified, the **on** struct member is implemented as a method, named **on_***event-type***()**. In this case, the action block can be invoked without the occurrence of the event, call the **on_***event-type***()** method. This method can be extended like any other method, by using **is**, **is also**, **is only**, or **is first** (see 18.1.3).

The following restrictions also apply:

— The named event shall be local to the struct specified by *const-exp*, or to the struct in which the **on** is defined if no *const_exp* is specified.

— The **on** action block shall not contain any time-consuming actions or TCMs.

See also 4.2.3 and 18.1.3.

Syntax example:

```
on xmit_ready {
    transmit()
}
```

13.3 on event-port

Purpose	Specify a block of actions that execute on on the triggering an event.	
Category	Struct member	
Syntax	**on** [*const-exp*.]*event-port-name***$ {***action***; ...}**	
Parameters	*const-exp*	An optional expression identifying the struct or unit in which event-type is defined. This expression must remain constant and thus can consist only of: — A **const** field — A unit instance name — Indexing a unit from a list of unit instances (a constant unit pointer) If not provided, the context struct or unit is assumed.
	event-port-name	The event port that invokes the action block. The port direction must be either **in** or **inout**.
	*action***; ...**	A block of non-time-consuming actions.

This defines a struct member that executes a block of actions immediately whenever a specified event port is triggered. An **on event-port** struct member is similar to a regular method, except that it is invoked automatically upon an occurrence of the event that triggers the event port. An **on** action block is executed before TCMs waiting for the same event. The actions are executed in the order in which they appear in the action block.

To extend an on struct member, its declaration can be repeated and use a different action block. This has the same effect as using is also to extend a method.

The following restriction also applies:

— The **on** action block shall not contain any time-consuming actions or TCMs.

See also 4.2.3 and 18.1.3.

Syntax example:

```
on sys.clk$ {
    out("clock tick")
}
```

13.4 expect | assume

Purpose	Define a temporal rule		
Category	Struct member		
Syntax	**expect** | **assume** [*rule-name* **is**] *temporal-expression* [**else dut_error(***string-exp***)**] [**using** *temporal_operators*] **expect** | **assume** *rule-name* [**is only** *temporal-expression* [**else dut_error(***string-exp***)**]] [**using also** *temporal_operators*]		
Parameters	*rule-name*	A name that uniquely identifies the rule from other rules or events within the struct. It can be used to override the temporal rule later on in the code or change from **expect** to **assume** or vice versa.	
	temporal-expression	A TE that is always expected to succeed. Typically involves a temporal yield (=>) operator (see 12.2.13).	
	string-exp	A string or an expression that can be converted to a string. If the TE fails, the string is printed.	
	temporal-operators	One or more temporal operators of the form: *operation condition*, separated by commas, where:	
		operation is one of the following keywords:	
		abort	Terminate evaluation of the temporal expression in the current tick and restart it in the next tick.
		start	Start the evaluation of the temporal expression in the current tick if the expect is currently in a stopped state.
		exclusive_ start	Similar to start, but in addition the expect is stopped when the struct is created, and remains stopped until acted upon by the exclusive start.
		stop	Stop the evaluation of the temporal expression in the current tick. It will remain stopped until a start instruction is issued.
		condition is one of the following:	
		@event-name	Triggering temporal event. This event can belong to the context struct (that is, [**me**.]event-name), or is reachable via a constant path (that is, *const-or-unit-instance-field-path.event-name*).
		none	Return to the default state. This value is intended for use when there was a previous triggering event for this instruction that should be removed.
		empty	Used to specify a dummy event (i.e., an event that never happens), for use with the procedural API - see (Yuri/AlanH: Add XREF to new section once done).

Both the **expect** and **assume** struct members are used for defining temporal properties.

— When a simulation-based tool executes the ***e*** program, it evaluates the rule expressed by the TE. If the TE fails at some point in time (see 12.3), the tool reports an error as specified with the **dut_error** clause (if no **dut_error** clause is specified, the tool prints the rule name). The notion of failure of the TE implies a new evaluation starts on every state following a state in which the sampling event

occurs (see 12.3.3). Simulation-based tools typically treat **expect** and **assume** in exactly the same manner.

— When a formal verification tool analyzes the ***e*** program, **expect** struct members are interpreted as rules the tool needs to verify, whereas **assume** struct members are interpreted as constraints on legal behavior. This means the tool looks for program execution paths where the TE bound to an **expect** fails (see 12.3) and none of the TEs bound to the **assume**s fail.

In addition, one or more temporal operators can also be attributed to it, with the **using** *temporal-operators* syntax.

Once a rule has been defined, it can be modified using the **is only** syntax and can be changed from an **expect** to an **assume** or vice versa. To perform multiple verification runs, the rules can be varied slightly or the same set of rules can be used in different **expect**/**assume** combinations. The **using also** *temporal-operators* syntax can be used to add a temporal operator in a redefinition of the rule (when used together with **is only**), or to add it as a separate extension to an existing rule (when used without **is only**).

Syntax example:

```
expect @a => {[1..5]; @b} @clk
```

Example

This example defines an **expect**, `bus_cycle_length`, that requires the length of the bus cycle to be no longer than 1000 cycles.

```
struct bus_e {
    event bus_clk        is change('top.b_clk') @sim;
    event transmit_start is rise  ('top.trans') @bus_clk;
    event transmit_end   is rise  ('top.transmit_done') @bus_clk;
    event bus_cycle_length;

    expect bus_cycle_length is
        @transmit_start => {[0..999]; @transmit_end} @bus_clk
    else dut_error("Bus cycle did not end in 1000 cycles")
}
```

13.5 Procedural API for Temporal Operators on event and expect struct Members

13.5.1 do_abort_on_event()

Purpose	Apply the abort operator on the event struct member	
Category	Predefined method of any struct or unit	
Syntax	**do_abort_on_event(***name***:** string**,** *force***:** bool**)**	
Parameters	*name*	The name of an event struct member of this struct or unit
	force	Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **abort** operator on the event name of the target struct, if attributed operators of the event *name* and *force* parameters allow it:

— If *force* is FALSE, the event *name* is affected only if no **abort** operator is attributed to it, or its condition is **none**; condition **@***event-name* or **empty** disables the method.

— If *force* is TRUE, the event *name* is affected regardless attributed operators.

Syntax example:

```
do_abort_on_event("checker", FALSE)
```

13.5.2 do_stop_on_event()

Purpose	Apply the stop operator on the event struct member	
Category	Predefined method of any struct or unit	
Syntax	**do_stop_on_event(***name***:** string**,** *force***:** bool**)**	
Parameters	*name*	The name of an event struct member of this struct or unit
	force	Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **stop** operator on the event name of the target struct, if attributed operators of the event *name* and *force* parameters allow it:

— If *force* is FALSE, the event *name* is affected only if no **stop** operator is attributed to it, or its condition is **none**; condition **@***event-name* or **empty** disables the method.

— If *force* is TRUE, the event *name* is affected regardless attributed operators.

Syntax example:

```
do_stop_on_event("checker", FALSE)
```

13.5.3 do_start_on_event()

Purpose	Apply the start operator on the event struct member	
Category	Predefined method of any struct or unit	
Syntax	**do_start_on_event(***name***:** string**,** *force***:** bool**)**	
Parameters	*name*	The name of an event struct member of this struct or unit
	force	Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **start** operator on the event name of the target struct, if attributed operators of the event *name* and *force* parameters allow it:

— If *force* is FALSE, the event *name* is affected only if no **start** operator is attributed to it, or its condition is **none**; condition **@***event-name* or **empty** disables the method.

— If *force* is TRUE, the event *name* is affected regardless attributed operators.

Syntax example:

```
do_start_on_event("checker", FALSE)
```

13.5.4 do_abort_on_expect()

Purpose	Apply the abort operator on the expect struct member	
Category	Predefined method of any struct or unit	
Syntax	**do_abort_on_expect(***name***:** string**,** *force***:** bool**)**	
Parameters	*name*	The name of an expect struct member of this struct or unit
	force	Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **abort** operator on the expect name of the target struct, if attributed operators of the expect *name* and *force* parameters allow it:

— If *force* is FALSE, the expect *name* is affected only if no **abort** operator is attributed to it, or its condition is **none**; condition **@***event-name* or **empty** disables the method.

— If *force* is TRUE, the expect *name* is affected regardless attributed operators.

Syntax example:

```
do_abort_on_expect("checker", FALSE)
```

13.5.5 do_stop_on_expect()

Purpose	Apply the stop operator on the expect struct member	
Category	Predefined method of any struct or unit	
Syntax	**do_stop_on_expect(***name***:** string**,** *force***:** bool**)**	
Parameters	*name*	The name of an expect struct member of this struct or unit
	force	Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **stop** operator on the expect name of the target struct, if attributed operators of the expect *name* and *force* parameters allow it:

— If *force* is FALSE, the expect *name* is affected only if no **stop** operator is attributed to it, or its condition is **none**; condition **@***event-name* or **empty** disables the method.

— If *force* is TRUE, the texpect *name* is affected regardless attributed operators.

Syntax example:

```
do_stop_on_expect("checker", FALSE)
```

13.5.6 do_start_on_expect()

Purpose	Apply the start operator on the expect struct member	
Category	Predefined method of any struct or unit	
Syntax	**do_start_on_expect(***name***:** string**,** *force***:** bool**)**	
Parameters	*name*	The name of an expect struct member of this struct or unit
	force	Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **start** operator on the expect name of the target struct, if attributed operators of the expect *name* and *force* parameters allow it:

— If *force* is FALSE, the expect *name* is affected only if no **start** operator is attributed to it, or its condition is **none**; condition **@***event-name* or **empty** disables the method.

— If *force* is TRUE, the expect *name* is affected regardless attributed operators.

Syntax example:

```
do_start_on_expect("checker", FALSE)
```

13.5.7 do_abort_on_struct()

Purpose	Apply the abort operator on events and expects of a struct. By default it applies it on all of them, but can be customized by extending the **apply_abort_on_struct()** method (see 13.5.10).	
Category	Predefined method of any struct or unit	
Syntax	**do_abort_on_struct(***force***:** bool**)**	
Parameters	*force*	Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **abort** operator on events and expects of the target struct, by calling the **apply_abort_on_struct()** method with the same *force* parameter. By default, its effect is as follows:

— If force is FALSE, the events and expects are affected only if no **abort** operator is attributed to them, or its condition is **none**; condition **@***event-name* or **empty** disables the method.

— If force is TRUE, the events and expects are affected regardless attributed operators.

Syntax example:

```
do_abort_on_struct(FALSE)
```

13.5.8 do_stop_on_struct()

Purpose	Apply the stop operator on events and expects of a struct. By default it applies it on all of them, but can be customized by extending the **apply_stop_on_struct()** method (see 13.5.11).
Category	Predefined method of any struct or unit
Syntax	**do_stop_on_struct(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **stop** operator on events and expects of the target struct, by calling the **apply_stop_on_struct()** method with the same *force* parameter. By default, its effect is as follows:

— If force is FALSE, the events and expects are affected only if no **stop** operator is attributed to them, or its condition is **none**; condition ***@****event-name* or **empty** disables the method.

— If force is TRUE, the events and expects are affected regardless attributed operators.

Syntax example:

```
do_stop_on_struct(FALSE)
```

13.5.9 do_start_on_struct()

Purpose	Apply the start operator on events and expects of a struct. By default it applies it on all of them, but can be customized by extending the **apply_start_on_struct()** method (see 13.5.12).
Category	Predefined method of any struct or unit
Syntax	**do_start_on_struct(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **start** operator on events and expects of the target struct, by calling the **apply_start_on_struct()** method with the same *force* parameter. By default, its effect is as follows:

— If force is FALSE, the events and expects are affected only if no **start** operator is attributed to them, or its condition is **none**; condition ***@****event-name* or **empty** disables the method.

— If force is TRUE, the events and expects are affected regardless attributed operators.

Syntax example:

```
do_start_on_struct(FALSE)
```

13.5.10 apply_abort_on_struct()

Purpose	Customize how the abort operator is applied on temporal struct members of a struct
Category	Predefined method of any struct or unit
Syntax	**apply_abort_on_struct(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

To customize how the **abort** operator is applied on temporal struct members, this method can be extended for the specific struct. The default implementation of the method contains two calls:

— do_abort_on_all_events(force);
— do_abort_on_all_expects(force);

In an extension, these or other predefined methods can be called, for example, **do_abort_on_event()** on a specific event.

Syntax example:

```
extend sys {
apply_abort_on_struct(force: bool) is only {
    do_abort_on_all_expects(force)
    }
}
```

13.5.11 apply_stop_on_struct()

Purpose	Customize how the stop operator is applied on temporal struct members of a struct
Category	Predefined method of any struct or unit
Syntax	**apply_stop_on_struct(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

To customize how the **stop**operator is applied on temporal struct members, this method can be extended for the specific struct. The default implementation of the method contains two calls:

— do_stop_on_all_events(force);
— do_stop_on_all_expects(force);

In an extension, these or other predefined methods can be called, for example, **do_stop_on_event()** on a specific event.

Syntax example:

```
extend sys {
apply_stop_on_struct(force: bool) is only {
    do_stop_on_all_expects(force)
    }
}
```

13.5.12 apply_start_on_struct()

Purpose	Customize how the start operator is applied on temporal struct members of a struct
Category	Predefined method of any struct or unit
Syntax	**apply_start_on_struct(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

To customize how the **start** operator is applied on temporal struct members, this method can be extended for the specific struct. The default implementation of the method contains two calls:

— do_start_on_all_events(force);
— do_start_on_all_expects(force);

In an extension, these or other predefined methods can be called, for example, **do_start_on_event()** on a specific event.

Syntax example:

```
extend sys {
apply_start_on_struct(force: bool) is only {
    do_start_on_all_expects(force)
    }
}
```

13.5.13 do_abort_on_all_events()

Purpose	Apply the abort operator on all events of a struct. This method is used in extensions of the **apply_abort_on_struct()** method (see 13.5.10), to customize how the abort operator is applied on temporal struct members a struct.
Category	Predefined method of any struct or unit
Syntax	**do_abort_on_all_events(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **abort** operator on all events of the struct, if attributed operators of the event and force parameter allow it:

— If force is FALSE, the events are affected only if no **abort** operator is attributed to them, or its condition is **none**; condition **@***event-name* or **empty** disables the method.
— If force is TRUE, the events are affected regardless attributed operators.

Syntax example:

```
extend sys {
apply_abort_on_struct(force: bool) is only {
    do_abort_on_all_events(force)
    }
```

```
}
```

13.5.14 do_stop_on_all_events()

Purpose	Apply the stop operator on all events of a struct. This method is used in extensions of the **apply_stop_on_struct()** method (see 13.5.11), to customize how the abort operator is applied on temporal struct members a struct.
Category	Predefined method of any struct or unit
Syntax	**do_stop_on_all_events(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **stop** operator on all events of the struct, if attributed operators of the event and force parameter allow it:

— If force is FALSE, the events are affected only if no **stop** operator is attributed to them, or its condition is **none**; condition **@***event-name* or **empty** disables the method.

— If force is TRUE, the events are affected regardless attributed operators.

Syntax example:

```
extend sys {
apply_stop_on_struct(force: bool) is only {
    do_stop_on_all_events(force)
    }
}
```

13.5.15 do_start_on_all_events()

Purpose	Apply the start operator on all events of a struct. This method is used in extensions of the **apply_start_on_struct()** method (see 13.5.12), to customize how the abort operator is applied on temporal struct members a struct.
Category	Predefined method of any struct or unit
Syntax	**do_start_on_all_events(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **start** operator on all events of the struct, if attributed operators of the event and force parameter allow it:

— If force is FALSE, the events are affected only if no **start** operator is attributed to them, or its condition is **none**; condition **@***event-name* or **empty** disables the method.

— If force is TRUE, the events are affected regardless attributed operators.

Syntax example:

```
extend sys {
apply_start_on_struct(force: bool) is only {
    do_start_on_all_events(force)
```

```
    }
}
```

13.5.16 do_abort_on_all_expects()

Purpose	Apply the abort operator on all expects of a struct. This method is used in extensions of the **apply_abort_on_struct()** method (see 13.5.10), to customize how the abort operator is applied on temporal struct members a struct.
Category	Predefined method of any struct or unit
Syntax	**do_abort_on_all_expects(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **abort** operator on all expects of the struct, if attributed operators of the expect and force parameter allow it:

— If force is FALSE, the expects are affected only if no **abort** operator is attributed to them, or its condition is **none**; condition **@***expect-name* or **empty** disables the method.

— If force is TRUE, the expects are affected regardless attributed operators.

Syntax example:

```
extend sys {
apply_abort_on_struct(force: bool) is only {
    do_abort_on_all_expects(force)
    }
}
```

13.5.17 do_stop_on_all_expects()

Purpose	Apply the stop operator on all expects of a struct. This method is used in extensions of the **apply_stop_on_struct()** method (see 13.5.11), to customize how the stop operator is applied on temporal struct members a struct.
Category	Predefined method of any struct or unit
Syntax	**do_stop_on_all_expects(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **stop** operator on all expects of the struct, if attributed operators of the expect and force parameter allow it:

— If force is FALSE, the expects are affected only if no **stop** operator is attributed to them, or its condition is **none**; condition **@***expect-name* or **empty** disables the method.

— If force is TRUE, the expects are affected regardless attributed operators.

Syntax example:

```
extend sys {
apply_stop_on_struct(force: bool) is only {
```

```
        do_stop_on_all_expects(force)
        }
    }
```

13.5.18 do_start_on_all_expects()

Purpose	Apply the start operator on all expects of a struct. This method is used in extensions of the **apply_start_on_struct()** method (see 13.5.12), to customize how the start operator is applied on temporal struct members a struct.
Category	Predefined method of any struct or unit
Syntax	**do_start_on_all_expects(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **start** operator on all expects of the struct, if attributed operators of the expect and force parameter allow it:

— If force is FALSE, the expects are affected only if no **start** operator is attributed to them, or its condition is **none**; condition **@***expect-name* or **empty** disables the method.

— If force is TRUE, the expects are affected regardless attributed operators.

Syntax example:

```
extend sys {
apply_start_on_struct(force: bool) is only {
    do_start_on_all_expects(force)
    }
}
```

13.5.19 do_abort_on_subtree()

Purpose	Apply the abort operator on the specified unit and propagate to the unit sub-tree under it.
Category	Predefined method of any unit
Syntax	**do_abort_on_subtree(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **abort** operator on events and expects of the target unit and unit sub-tree under it, by calling the **propagate_abort_on_subtree()** method with the same *force* parameter. By default, its effect is as follows:

— If force is FALSE, the events and expects are affected only if no **abort** operator is attributed to them, or its condition is **none**; condition **@***event-name* or **empty** disables the method.

— If force is TRUE, the events and expects are affected regardless attributed operators.

This default effect can be changed by extending the **propagate_abort_on_subtree()** method (see 13.5.22).

Syntax example:

```
do_abort_on_subtree(FALSE)
```

13.5.20 do_stop_on_subtree()

Purpose	Apply the stop operator on the specified unit and propagate to the unit sub-tree under it.
Category	Predefined method of any unit
Syntax	**do_stop_on_subtree(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **stop** operator on events and expects of the target unit and unit sub-tree under it, by calling the **propagate_stop_on_subtree()** method with the same *force* parameter. By default, its effect is as follows:

— If force is FALSE, the events and expects are affected only if no **stop** operator is attributed to them, or its condition is **none**; condition ***@****event-name* or **empty** disables the method.

— If force is TRUE, the events and expects are affected regardless attributed operators.

This default effect can be changed by extending the **propagate_stop_on_subtree()** method (see 13.5.23).

Syntax example:

```
do_stop_on_subtree(FALSE)
```

13.5.21 do_start_on_subtree()

Purpose	Apply the start operator on the specified unit and propagate to the unit sub-tree under it.
Category	Predefined method of any unit
Syntax	**do_start_on_subtree(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method immediately applies the **start** operator on events and expects of the target unit and unit sub-tree under it, by calling the **propagate_start_on_subtree()** method with the same *force* parameter. By default, its effect is as follows:

— If force is FALSE, the events and expects are affected only if no **start** operator is attributed to them, or its condition is **none**; condition ***@****event-name* or **empty** disables the method.

— If force is TRUE, the events and expects are affected regardless attributed operators.

This default effect can be changed by extending the **propagate_start_on_subtree()** method (see 13.5.24).

Syntax example:

```
do_start_on_subtree(FALSE)
```

13.5.22 propagate_abort_on_subtree()

Purpose	Customize how the abort operator is propagated on temporal struct members of a specified unit and the unit sub-tree under it
Category	Predefined method of any unit
Syntax	**propagate_abort_on_struct(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

To customize how the **abort** operator is propagated on temporal struct members of a specified unit and the unit sub-tree under it, this method can be extended for the specific unit. The default implementation of the method contains two calls:

— do_abort_struct(force);
— do_abort_on_all_instance_fields(force);

In an extension, these or other predefined methods can be called, for example, **do_abort_on_struct()** can be called on a struct pointed to by a specific field.

Syntax example:

```
extend sys {
propagate_abort_on_subtree(force: bool) is only {
        do_abort_on_all_instance_fields(force)
    }
}
```

13.5.23 propagate_stop_on_subtree()

Purpose	Customize how the stop operator is propagated on temporal struct members of a specified unit and the unit sub-tree under it
Category	Predefined method of any unit
Syntax	**propagate_stop_on_struct(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

To customize how the **stop** operator is propagated on temporal struct members of a specified unit and the unit sub-tree under it, this method can be extended for the specific unit. The default implementation of the method contains two calls:

— do_stop_struct(force);
— do_stop_on_all_instance_fields(force);

In an extension, these or other predefined methods can be called, for example, **do_stop_on_struct()** can be called on a struct pointed to by a specific field.

Syntax example:

```
extend sys {
propagate_stop_on_subtree(force: bool) is only {
```

```
            do_stop_on_all_instance_fields(force)
        }
    }
```

13.5.24 propagate_start_on_subtree()

Purpose	Customize how the start operator is propagated on temporal struct members of a specified unit and the unit sub-tree under it
Category	Predefined method of any unit
Syntax	**propagate_start_on_struct(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

To customize how the **start** operator is propagated on temporal struct members of a specified unit and the unit sub-tree under it, this method can be extended for the specific unit. The default implementation of the method contains two calls:

— do_start_struct(force);
— do_start_on_all_instance_fields(force);

In an extension, these or other predefined methods can be called, for example, **do_start_on_struct()** can be called on a struct pointed to by a specific field.

Syntax example:

```
extend sys {
propagate_start_on_subtree(force: bool) is only {
        do_start_on_all_instance_fields(force)
    }
}
```

13.5.25 do_abort_on_all_instance_fields()

Purpose	Propagate the abort operator on all instance fields of a unit. This method is used in extensions of the **propagate_abort_on_subtree() method** (see 13.5.22), to customize how the abort operator is propagated on temporal struct members of a unit and unit sub-tree under it
Category	Predefined method of any unit
Syntax	**do_abort_on_all_instance_fields(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method recursively applies the **abort** operator on on all instance fields of the unit, by calling the **do_abort_on_subtree()** method on them with the same *force* parameter.

Syntax example:

```
extend sys {
    propagate_abort_on_subtree(force: bool) is only {
        do_abort_on_all_instance_fields(force)
```

```
    }
}
```

13.5.26 do_stop_on_all_instance_fields()

Purpose	Propagate the stop operator on all instance fields of a unit. This method is used in extensions of the **propagate_stop_on_subtree() method** (see 13.5.23), to customize how the stop operator is propagated on temporal struct members of a unit and unit sub-tree under it
Category	Predefined method of any unit
Syntax	**do_stop_on_all_instance_fields(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method recursively applies the **stop** operator on on all instance fields of the unit, by calling the **do_stop_on_subtree()** method on them with the same *force* parameter.

Syntax example:

```
extend sys {
    propagate_stop_on_subtree(force: bool) is only {
        do_stop_on_all_instance_fields(force)
    }
}
```

13.5.27 do_start_on_all_instance_fields()

Purpose	Propagate the start operator on all instance fields of a unit. This method is used in extensions of the **propagate_start_on_subtree() method** (see 13.5.24), to customize how the start operator is propagated on temporal struct members of a unit and unit sub-tree under it
Category	Predefined method of any unit
Syntax	**do_start_on_all_instance_fields(***force***:** bool**)**
Parameters	*force* Denotes whether the effect is forced, regardless attributed operators

This method recursively applies the **start** operator on on all instance fields of the unit, by calling the **do_start_on_subtree()** method on them with the same *force* parameter.

Syntax example:

```
extend sys {
    propagate_start_on_subtree(force: bool) is only {
        do_start_on_all_instance_fields(force)
    }
}
```

