

11. Time-consuming actions 1

Time-consuming actions are distinguished actions that shall only appear in the body of a TCM (see Clause 15); however, they shall not appear in any of the following contexts, even if they are themselves embedded within a TCM: 5

- **exec** action block (see 9.2.19)
- **new ..** with action block (see 2.15.2)
- **error()** action block (see 14.3.2) 10

Any attempt to use a time-consuming action in an illegal context shall cause the compiler to issue an error message.

11.1 Synchronization actions 15

The **sync** (see 11.1.2) and **wait** (see 11.1.3) actions are used to synchronize temporal test activities within an *e* program, and between the DUT and the *e* program. Both actions operates on an optional temporal expression (see Clause 9); if it is omitted, then the temporal expression cycle is implied (see Clause 9). 20

11.1.1 Synchronization semantics

The declaration of a TCM introduces a default sampling event (see Clause 15), which serves as the sampling event for any temporal expression embedded in a **wait** or **sync** action that does not have its own sampling event specified. Thus the following examples are equivalent 25

```
tcm() @p_clk is { wait };
tcm() @p_clk is { wait cycle@p_clk };
```

A number of concurrent TCMs can be invoked using the **start** action (see 15.2.2). An *e* program consisting of a number of concurrently executing TCMs simulates concurrency by interleaving their actions over time. The temporal expressions embedded in the synchronization actions orchestrate this behavior as follows. 30

- a) When an executing TCM reaches a synchronization action, its execution shall be suspended and some other suspended TCM can then be scheduled for execution. 35
- b) When a suspended TCM is scheduled for execution, the system first evaluates the embedded temporal expression:
 - 1) if this succeeds, the TCM resumes execution immediately;
 - 2) if it remains undecided, the TCM remains suspended;
 - 3) if it fails, the TCM remains idle until the next occurrence of the sampling event. 40

When a TCM resumes, it commences executing from the first action following the synchronization action, and continues execution without interruption until it either terminates normally (for a started TCM) or encounters a subsequent synchronization action. 45

In this way, the synchronization actions in the body of a TCM delimit the program fragments that [are](#) executed atomically without interruption; the system can pause and choose another TCM for execution only when a synchronization action is encountered. 50

The following restrictions also apply. 50

- If a called TCM returns, the callee resumes execution immediately.
- If one TCM calls another, there is an implicit “sync cycle” upon commencing to execute the called TCM. 55

11.1.2 sync

Purpose	Synchronize a TCM to the success of the embedded temporal expression.
Category	Action
Syntax	sync [<i>temporal-expression</i>]
Parameters	<i>temporal-expression</i> A temporal expression that specifies how the synchronization is achieved.

When an TCM reaches a **sync** action, its execution shall be suspended so some other TCM, or the suspended TCM itself, can be scheduled for execution.

If the *temporal-expression* is omitted, `cycle` is assumed and the TCM therefore synchronizes to its default sampling event.

Syntax example:

```
sent_data_tcm();
sync;
```

11.1.3 wait

Purpose	Wait until a temporal expression succeeds
Category	Action
Syntax	wait [[until] <i>temporal-expression</i>]
Parameters	<i>temporal-expression</i> A temporal expression that specifies how the synchronization is achieved.

When a TCM reaches a **wait** action, its execution shall be suspended so some other TCM can be scheduled for execution. The suspended TCM itself will remain idle and will not be scheduled until the next occurrence of the sampling event.

If the *temporal-expression* is omitted, `cycle` is assumed and the TCM therefore waits for the next occurrence of the default sampling event.

NOTE—The **until** option is an aid to readability only and has no effect on the semantics of the action.

Syntax example:

```
wait [3]*cycle;
```

11.2 Concurrency actions

The **all of** (see 11.2.1) and **first of** (see 11.2.2) actions facilitate concurrent execution within TCMs. They introduce explicit concurrent branches whose actions shall be interleaved in the same way that the actions of concurrent TCMs are themselves interleaved. The parallel action blocks that form the body of the **all of** and

first of actions can start or call TCMs, synchronize through **wait** and **sync** actions, and execute all normal and time consuming actions subject to the following limitations. 1

- It is illegal to execute a **return** action (see 15.2.5) from the body of a concurrency action.
- It is illegal to execute either a **break** or **continue** action (see 18.4) from the body of a concurrency action unless the loop itself is embedded in the body of the concurrency action. 5

The compiler shall issue an error message in each of the above circumstances. 10

11.2.1 all of 10

Purpose	Execute action blocks in parallel	15
Category	Action	
Syntax	all of {{ <i>action</i> ; ...}; ... }	
Parameters	{ <i>action</i> ; ...}; Action blocks that are to execute concurrently. Each action block is a separate branch.	20

This executes multiple action blocks concurrently as separate branches of a fork, continuing with subsequent actions only when all the action blocks have completed executing. 25

When execution of a TCM reaches an **all of** action, the branches shall be executed in some indeterminate order until each branch either terminates normally, or reaches a synchronization action (see 11.1) and is suspended. The rules for scheduling and resuming the execution of the branches of an **all of** action are the same as those for concurrent TCMs (see 11.1.1). 30

The subsequent actions following the **all of** action shall be reached when and only when execution of all the individual branches in the body has completed. When the last branch of the **all of** action completes, the subsequent actions commence execution immediately. 35

Syntax example: 35

```
all of {
    {check_bus_controller();};
    {check_memory_controller();};
    {wait cycle; check_alu();};
};
```

40

45

50

55

11.2.2 first of

Purpose	Execute action blocks in parallel
Category	Action
Syntax	first of { <i>action</i> ; ...}; ... }
Parameters	{ <i>action</i> ; ...}; Action blocks that are to execute concurrently. Each action block is a separate branch.

This executes multiple action blocks concurrently as separate branches of a fork, continuing with subsequent actions once one of the action blocks has completed executing.

When execution of a TCM reaches a **first of** action, the branches shall be executed in some indeterminate order until some branch terminates normally, or each branch reaches a synchronization action (see 11.1) and is suspended. The rules for scheduling and resuming the execution of the branches of a **first of** action are the same as those for concurrent TCMs (see 11.1.1).

The subsequent actions following the **first of** action shall be reached when and only when execution of one of the individual branches in the body has completed. When one of the branches of the **first of** action completes, the subsequent actions commence execution immediately and the suspended parallel branches of the **first of** action are terminated (or preempted) immediately.

Syntax example:

```
first of { {wait [3]*cycle@ev_a}; {wait @ev_e; }; };
```

11.3 State machines

This section describes the syntax and semantics of the state machine time-consuming action. The state machine action is used to in-line a finite state machine in the body of a time-consuming method. *See also:* Clause 15.

11.3.1 Overview

The **state machine** action is a construct for modeling finite state machines. A *state machine definition* consists of the state machine action block together with the identification of a state variable that holds the current state of the machine. The *state variable* is a local variable (see Clause 2) of the enclosing TCM or a field path expression (****Clause ??**); in either case, the entity shall be a scalar of some enumerated type (see Clause 2). The symbolic values of the enumerated type are the states of the state machine being defined. The rules governing state variables are detailed in 11.3.2.

The **state machine** action block specifies the behavior of the state machine. This action block consists of a number of special state-transition actions and state actions. A *state-transition action* is a time-consuming action that governs how the state machine changes from one state to another. There are two kinds of state-transition action, as detailed in 11.3.3 and 11.3.4. A **state action** is a block that is executed whenever the identified state of the machine is entered (see 11.3.5).

In addition, the **state machine** action can nominate a final state of the state machine. If it does so, when the machine enters this state, the **state machine** action shall terminate and subsequent actions shall commence execution.

11.3.2 state machine action

Purpose	Define a state machine
Category	Action
Syntax	state machine <i>state-variable</i> [until <i>final-state</i>] {(<i>state-transition-action</i> <i>state-action</i>); ...}
Parameters	<i>state-variable</i> This is a local variable (or a field) of some enumerated type.
	<i>final-state</i> The state at which the state machine normally terminates.
	<i>state-transition-action</i> A state transition, which occurs when the associated action block finishes. See 11.3.3 and 11.3.4.
	<i>state-action</i> Actions executed upon entering the named state (see 11.3.5).

This action declares a state machine using a *state-variable* of some enumerated type to hold the current state of the machine. The states of the machine are the symbolic values of this variable's enumerated type. When the **state machine** action is reached, the state machine starts in the first state listed in the state variable's enumerated type. If the optional **until** *final-state* exit condition is used, the state machine runs until that state is reached, whereupon it terminates. *final-state* shall be one of the values of the state-variable's enumerated type. If the **until** clause is not used, the state machine runs until the enclosing construct, usually a TCM action block or a concurrency action, is terminated in some way.

The definition of the *state machine* consists of a number of state-transition actions and state actions:

- a) Simple state transition action: `state => state {action; ...}`
- b) Wildcard state transition action: `*=> state {action; ...}`
- c) State action: `=> state {action; ...}`

In each case, the state shall be one of the legal enumerated values of the *state-variable* used in the definition of the state machine. The action block associated with the *state* or *state-transition* actions can contain any legal actions, time-consuming or otherwise, subject to the following limitations.

- It is illegal to use a **return** action (see 15.2.5) in a *state* or *state-transition* action block.
- It is illegal to use a **break** or **continue** action (see 18.4) in a *state* or *state-transition* action block, unless enclosed by a **loop** action within the block.

The compiler shall issue an error message in each of the above circumstances.

The effect of assigning to the *state-variable* used in the definition of the state machine in any of the *state* or *state-transition* action blocks is undefined. Nested state machines shall use distinct *state* variables to hold their state; otherwise, the result is undefined.

Syntax example:

```
var stv : [initial, running, done];
state machine stv until done {
  initial => running { wait until rise('top.a') };
  initial => done { wait until change('top.r1'); wait until
    rise('top.r2') };
  running => initial { wait until rise('top.b') };
  running {
```

```

1         out("Entered ",stv," state at ",sys.time);
           while TRUE {wait cycle; out("still running");}
           };
5     };

```

11.3.3 Simple state transition: state => state

Purpose	One-to-one state transition	
Category	State transition	
Syntax	<i>current-state</i> => <i>next-state</i> { <i>action</i> ; ...}	
Parameters	<i>current-state</i>	The state from which the transition starts.
	<i>next-state</i>	The state to which the transition changes.
	<i>action</i> ; ...	The sequence of actions that precede the transition.

This specifies how a transition occurs from one state to another. The action block starts executing when the state machine enters the current state. When the action block completes, the transition to the next state occurs. The action block usually contains at least one time-consuming action; if it does not, the transition to the next state is immediate.

If the *current-state* is applicable to two or more simple or wildcard *state-transition* actions, then each action block commences execution concurrently when the *current-state* is entered; in this case, the actual next state chosen is indeterminate and depends upon the first of the action blocks to complete execution. When the state transition occurs, any concurrent action blocks associated with the *current-state* shall be preempted.

Syntax example:

```

begin => run {wait [2]*cycle; out("Run state entered")};

```

11.3.4 Wildcard state transition: * => state

Purpose	Any-to-one state transition	
Category	State transition	
Syntax	* => <i>next-state</i> { <i>action</i> ; ...}	
Parameters	<i>next-state</i>	The state to which the transition changes.
	<i>action</i> ; ...	The sequence of actions that precede the transition.

This specifies how a transition occurs from any defined state to a particular state. The action block starts executing when the state machine enters a new state. When the action block completes, the transition to the next state occurs.

If two or more simple or wildcard *state-transition* actions apply to any state of the machine, then each action block commences execution concurrently when that state is entered; in this case, the actual next state chosen is indeterminate and depends upon the first of the action blocks to complete execution. When the state transition occurs, any concurrent action blocks associated with that state shall be preempted.

Syntax example:

```
* => pause {wait @sys.restart; out("Entering pause state");};
```

11.3.5 state action

Purpose	Execute actions upon entering a state, with no state transition	
Category	State action block	
Syntax	<i>current-state</i> { <i>action</i> ; ...}	
Parameters	<i>current-state</i>	The state for which the action block is to be executed.
	<i>action</i> ; ...	The sequence of actions that is executed upon entering the current state.

This specifies an action block that is executed when a specific state is entered. No transition occurs when the action block completes.

- If the *current-state* is also the final state of the state machine then the state machine terminates when the action block completes and any actions subsequent to the state machine action shall then commence execution immediately.
- There may be more than one state action associated with the *current-state*; in this case, all the action blocks commence execution concurrently when the *current-state* is entered.
- If the *current-state* has one or more *state-transition* actions associated with it then the state action block and the relevant *state-transition* actions all commence execution concurrently when the *current-state* is entered. If some *state-transition* action block completes execution, the state transition occurs immediately, possibly preempting the state action block if it has not already completed execution.

Syntax example:

```
* => run {out("* to run"); wait cycle};
run {out("In run state"); wait cycle; out("Still in run");};
run => done {out("run to done"); wait cycle};
```