9. *e* ports

This clause describes ports, ***e*** unit members that enhance the portability and interoperability of verification environments by making separation between an ***e*** unit and its interface possible.

9.1 Introduction to *e* ports

A *port* is an ***e*** unit member that makes a connection between an ***e*** unit and its interface to another internal or external entity. There are two ways to use ports:

— Internal ports (***e***2***e*** ports) connect an ***e*** unit to another ***e*** unit.

— External ports connect an ***e*** unit to a simulated object.

External ports are a generic way to access simulated objects of various kinds. An external port is bound to a simulated object, e.g., an HDL signal in the DUT. Then all access to that signal is made via the port. The port can be used to access a different signal simply by changing the binding; all the code that reads or writes to the port remains the same. Similarly, port semantics remain the same, regardless of what simulator is used. A *simulator* is any hardware or software agent that runs in parallel with an ***e*** program and models the behavior of any part of the DUT or its environment.

9.1.1 Creating port instances

A *port type* is defined by the following aspects:

a) The kind of port: simple port, buffer port, event port, or method port.
   1) *Simple ports* access data directly.
   2) *Buffer ports* implement an abstraction of queues, with blocking **get()** and **put()**.
   3) *Event ports* transfer events between ***e*** units or between an ***e*** unit and a simulator.
   4) *Method ports* enable a regular or TCM defined in an ***e*** unit or a foreign programming language module to be called from another ***e*** unit or foreign programming language module.

b) Direction, either input or output (or inout for simple and event ports).

c) Data element, the ***e*** type that can be passed through this port.

Ports can only be instantiated within units using a unique instance name and the port type (direction, port kind, and a kind-specific type specifier). Like units, port instances are generated during pre-run generation and cannot be created, modified, or removed during a run.

The generic syntax for ports is:

port-instance-name **:** [*direction*] *port-kind* [**of** *type-specifier*] **is instance**

Event ports do not have a type specifier.

Examples

The following unit member creates a port instance:

```
data_in : in buffer_port of packet is instance
```

where

— The port instance name is `data_in`.

— The port kind is a buffer port.

— The port direction is input.

— The data element the port accepts is `packet`.

As another example, the following line creates a list of simple ports that each pass data of type `bit`:

```
ports : list of simple_port of bit is instance
```

9.1.2 Using ports

A port's behavior is influenced by port attributes, such as **hdl_path()** or **bind()**, that are applied to port instances using pre-run generation **keep** constraints. For example, the following lines of code create a port named `data` and connect (bind) it to an external simulator-related object whose HDL pathname is `data`.

```
data : inout simple_port of list of bit is instance;
  keep bind(data, external);
  keep data.hdl_path() == "data"
```

Each port kind has predefined methods that can be used to access the port values. For example, buffer ports have a predefined method **put()**, which writes a value onto an output port, as follows:

```
data_out : out buffer_port of cell is instance;

drive_all() @sys.any is {
    var stimuli : cell;
    var counter : int = 0;

    while counter < cells do {
        wait [1]*cycle;
        gen stimuli;
        data_out.put(stimuli);
        counter += 1
    }
}
```

9.1.3 Using port values and attributes in constraints

Like units, port instances can be created only during pre-run generation. They cannot be created by using **new** or generated at runtime. Consequently, a port value cannot be initialized or sampled in pre-run generation constraints. Port values can be used in on-the-fly generation constraints, in accordance with the basic constraint principles, such as the bidirectional nature of constraints.

9.2 Using simple ports

Simple ports can be used to transfer one data element at a time to or from an external simulated object, such as a Verilog register, a VHDL signal, a SystemC field, or an internal object (another ***e*** unit). A simple port's direction can be either input, output, or inout.

Use the **$** port access operator to read or write port values. To access MVL on simple ports, either declare a port's data element to be mvl or list of mvl, or use the MVL methods. See 9.2.1 and 9.2.2 for more information.

Internal and external ports shall have a **bind()** attribute that defines how they are connected. In addition, the **delayed()** attribute can be used to control whether new values are propagated immediately or at the next tick.

An external simple port needs to have an **hdl_path()** attribute to specify the name of the object to which it is connected. In addition, an external simple port can have several additional attributes that enable continuous driving of external signals (see 9.7).

9.2.1 Accessing simple ports and their values

Ports are containers, and the values they hold are separate entities from the port itself. The **$** access operator distinguishes port value expressions from port reference expressions.

The **$** operator, e.g., `p$`, can also be used to access or update the value held in a simple port `p`. When used on the RHS, `p$` refers to the port's value. On the LHS of an assignment, `p$` refers to the value's location, so an assignment to `p$` changes the value held in the port.

Without the **$** operator, an expression of any type port refers to the port itself, not to its value. In particular, an expression without the **$** operator can be used for operations involving port references.

Examples

Accessing port values

`print p$`	Prints the value of a simple port, `p`. [a]
`p$ = 0`	Assigns the value `0` to a simple port, `p`. [b]
`force p$= 0`	Forces a simple external port to `0`.
`print q$[1:0]`	Prints the two lists of the value of `q`.

[a]Compare with `print p`, which prints information about port `p`.

[b]Compare `p$ = 0;` with `pref = NULL`, which modifies a port reference so it does not point to any port instance.

Accessing a port

`print p`	Prints the information about port `p`. Port `p` is defined as: `p: simple_port of int (bits:8) is instance`
`keep q == p`	`q` refers to the port instance `p`. Port reference `q` is defined as: `!q: simple_port of int (bits:8)`

9.2.2 MVL on simple ports

There are two ways to read and write MVL on simple ports, as follows:

— Define a port and use the predefined MVL methods described in 9.9 to read and write values to the port.

— Define ports of type mvl or list of mvl and use the **$** access operator to read and write the port values.

Ports of type mvl or list of mvl (MVL ports) allow easy transformation between exact ***e*** values and MVL, which is useful for communicating with objects that sometimes model bit values other than `0` or `1` during a test. Otherwise, using non-MVL ports is preferable, since they allow keeping the port values in a bit-by-bit representation, while MVL ports require having an ***e*** list for an MVL vector. MVL type definition and MVL functions are described in 9.9.

The Verilog comparison operators (=== or !==) cannot be used with numeric ports or MVL ports. These operators can be used only with the tick access syntax.

9.2.3 @sim temporal expressions with external simple ports

Specifying an event port causes ***e*** to be sensitive to the corresponding HDL signal during the entire simulation session. This might result in some unnecessary runtime performance cost if ***e*** only needs to be sensitive in certain scenarios. In such cases, use an external simple port in TEs with **@sim** instead. The syntax is:

[**change** | **rise** | **fall**] **(***simple-port***$)@sim**

Typically, this syntax is used in wait actions.

Example

```
transaction_complete : in simple_port of bit is instance;
  keep bind(transaction_complete, external);

write_transaction(data: list of byte) @clk$ is {
    //...
    data_port$ = data;
    wait rise(transaction_complete$)@sim
}
```

Trying to apply the **@sim** operator to a bound internal port shall cause an error when the corresponding TE is evaluated, which occurs at runtime.

9.3 Using buffer ports

Buffer ports can be used to insert data elements into a queue or extract elements from a queue. Data is inserted and extracted from the queue in first-in-first-out (FIFO) order. When the queue is full, write-access to the port is blocked. When the queue is empty, read-access to the port is blocked. The queue size is fixed during generation by the **buffer_size()** attribute and cannot be changed at runtime. The queue size can be set to 0 for rendezvous ports. See 9.7.2.2 and 9.3.1 for more information.

A buffer port's direction can be either input or output. Use the buffer port's predefined **get()** and **put()** methods to read or write port values. These methods are *time-consuming methods* (TCMs). The **$** port access operator cannot be used with buffer ports.

Buffer ports shall have a **bind()** attribute that defines how they are connected. In addition, the **delayed()** attribute can be used to control whether new values are propagated immediately or at the next tick. The **pass_by_pointer()** attribute controls how data elements of composite type are passed. See also 9.7.

9.3.1 Rendezvous-zero size buffer queue

In rendezvous-style handshaking protocol, access to a port is blocked after each **put()** until a subsequent **get()** is performed, and access is blocked after each **get()** until a subsequent **put()** is performed.

This style of communication is easily achieved by using buffer ports with a data queue size of 0. The following example shows how this is done.

Example

```
unit consumer {
    in_p : in buffer_port of atm_cell is instance
}
unit producer {
    out_p : out buffer_port of atm_cell is instance
};
extend sys {
    consumer : consumer is instance;
    producer : producer is instance;
    keep bind(producer.out_p, consumer.in_p);
    keep producer.out_p.buffer_size() == 0
}
```

9.4 Using event ports

Event ports can be used to transfer events between two ***e*** units or between an ***e*** unit and an external object. An internal event port's direction can be either input, output, or inout. Use the **$** port access operator to read or write port values (see 9.4.1).

Internal and external ports need to have a **bind()** attribute that defines how they are connected. An external port needs to have an **hdl_path()** attribute to specify the name of the object to which it is connected. The **edge()** attribute for an external input event port specifies the edge on which an event is generated. See also 9.7.

9.4.1 Accessing event ports

Use the **$** access operator to access the event associated with an event port. An expression of type **event_port** without the **$** operator refers to the port itself and not to its event.

Example

This example shows how to connect event ports [using a **bind()** constraint] and use the **$** operator to access event ports in event contexts.

```
unit u1 {
    in_ep : in event_port is instance;
    tcm1()@in_ep$ is {
        // ...
    }
};

unit u2 {
    out_ep  : out event_port is instance;
    counter : uint;
    event clk is @sys.any;

    on clk {
        counter = counter + 1;
        if counter %10 == 0 then {
            emit out_ep$
        }
    }
};

extend sys {
    u1 : u1 is instance;
    u2 : u2 is instance;
      keep bind(u1.in_ep, u2.out_ep)
}
```

9.5 Using method ports

Method ports can be used to either call or export methods and TCMs defined in other ***e*** units or in foreign programming language modules. The advantages of method ports are:

— A transaction-level interface can be implemented between ***e*** and a high-level model described in a foreign language.

— The decision about which method to call (e.g., an ***e*** method or a foreign function) can be postponed from compile time to pre-run generation.

9.5.1 Method types

A method port shall be parameterized by a type of a special kind—a method type. The *method type* specifies the prototype (signature) of the method and implies specific user-defined semantics. For example, the following declares a method type for a method that accepts two integer arguments and returns an integer:

```
method_type adder_method_t(arg1:int, arg2:int): int
```

The following method type declaration has the same prototype as adder_method_t, but implies different user-defined semantics:

```
method_type local_adder_method_t(arg1:int, arg2:int): int
```

A method type that is associated with a TCM shall be defined with the **@sys.any** sampling event, e.g.,

```
method_type send_packet_method_t(p:packet)@sys.any
```

Method types shall be defined with a unique name; this name shall be explicitly specified in the instance declaration of the method port (see 9.6.4). For example, the following associates the `add` method port with the adder_method_t method type:

```
add : out method_port of adder_method_t is instance
```

The method type has semantic implications for a port beyond the simple matching of parameters and result types; it is also used to clarify runtime messages related to a particular method port. Thus, two method ports cannot be bound just because they have the same signature; they also need to be associated with the same method type.

9.5.2 Input method ports

An *input method port* declares an ***e*** method as callable from another ***e*** unit or from a foreign agent. The method port instance shall:

a) Reside in the same unit as its associated method;

b) Have an instance name that matches the name of the associated method;

c) Have a method type that matches the prototype of the associated method.

The method type and its prototype match if:

1) They have the same number of parameters.

2) Any parameters are of the same types (and in the right order).

3) Any return values are of the same type.

d) Include the **@sys.any** sampling event (in the method type declaration) if the method type is associated with a TCM.

9.5.3 Output method ports

Output method ports can be used to call regular or time-consuming methods defined in other ***e*** units or written in foreign programming languages.

9.5.4 Invoking method ports

The **$()** access operator can be used to call the method port (see also 9.6.13). The rules for parameter type checking, TCM call requirements, etc., are the same as those for invoking an ***e*** method directly. In particular, TCM method ports can only be called from inside a TCM scope.

The parameter passing semantics are the same as in direct calls to ***e*** methods. Scalar parameters are passed by value, while composite parameters (struct or list types) are passed by reference.

Other considerations:

— Do not rely on the ability to modify separate fields or list elements of the incoming parameter in the actual method. Instead, use the return value (or explicit passing of parameters by reference).

— All ports are elaborated after the end of **post_generate()**. Thus, method ports cannot be invoked before generation or from constraints.

— Calling an empty-bound method port is equivalent to calling an empty ***e*** method.

9.5.5 Binding method ports

If a set of input and output ports are bound, all the ports are connected (no matter how the binding pairs were specified) and a change on any output port affects all input ports. While this makes sense for simple ports, which are used to emulate wires, it does not for method ports. For example, if there are two output method ports, `Ao` and `Bo`, three input method ports, `Ai`, `Bi`, and `ABi`, and the binding looks like:

```
bind(Ao, Ai);
bind(Bo, Bi);
bind(Ao, ABi);
bind(Bo, ABi)
```

the intention probably is that a call to `Ao` causes a call of `Ai` and `ABi`, while a call to `Bo` causes a call of `Bi` and `ABi`. This intention is implemented; however, a call to `Ao` also causes a call of `Bi`, and a call to `Bo` also causes a call of `Ai`.

To bind multiple output ports to a common input, define the common input as a list of in method ports (see 9.6.4). Then, each of the input method ports is associated with the method via the list name.

Example

The list of in method ports is

```
type src_t : [A, B];
method_type p_t(s:src_t);

extend sys {
    Ao : out method_port of p_t is instance;
    Bo : out method_port of p_t is instance;

    ABi : list of in method_port of p_t is instance;
      keep ABi.size() == 2;

    ABi(src: src_t) is {
        out("AB(", src, ")")
    }
```

and the binding is:

```
// each output also invokes the common input
      keep bind(Ao, ABi[0]);
      keep bind(Bo, ABi[1]);

    run() is also {
        Ao$(A);
        Bo$(B)
    }
}
```

9.6 Defining and referencing ports

This subclause details how to define or reference a port.

9.6.1 simple_port

Purpose	Access other port instances or external simulated objects directly	
Category	Unit member	
Syntax	*port-instance-name* **:** [**list of**] [*direction*] **simple_port of** *element-type* **is instance**	
Parameters	*port-instance-name*	A unique identifier used to reference the port or access its value.
	direction	One of **in**, **out**, or **inout**. The default is **inout**, which means values can be read from and written to this port. For an **in** port, values can only be read from the port; for an **out** port, values can only be written to the port.
	element-type	Any predefined or user-defined ***e*** type, except a port type or unit type.

Simple ports can be used to transfer one data element at a time to or from an external simulated object or internal object (another ***e*** unit). External ports can transfer scalar types and lists of scalar types, including MVL data elements. Structs or lists of structs cannot be passed through external simple ports.

The port can be configured to access a different signal simply by changing the binding; all the code that reads or writes to the port remains the same. Similarly, port semantics remain the same, regardless of what simulator is used. Binding is fixed during generation.

A simple port's direction can be either in, out, or inout. Omitting the direction is the same as writing inout. Port types with different directions are not equivalent. The following types are fully equivalent:

```
data :       simple_port of byte is instance;
data : inout simple_port of byte is instance
```

Syntax example:

```
data : in simple_port of byte is instance
```

9.6.2 buffer_port

Purpose	Implement an abstraction of queues with blocking get and put	
Category	Unit member	
Syntax	*port-instance-name* **:** [**list of**] *direction* **buffer_port of** *element-type* **is instance**	
Parameters	*port-instance-name*	A unique identifier used to reference the port or access its value.
	direction	One of **in** or **out**. There is no default. For an **in** port, values can only be read from the port; for an **out** port, values can only be written to the port. See 9.8 for information on how to read and write buffer ports.
	element-type	Any predefined or user-defined ***e*** type, except a port type or a unit type.

Buffer ports can be used to insert data elements into a queue or extract elements from a queue. Data is inserted and extracted from the queue in FIFO order. When the queue is full, write-access to the port is blocked. When the queue is empty, read-access to the port is blocked.

The queue size is fixed during generation by the **buffer_size()** attribute and cannot be changed at runtime. The queue size can be set to `0` for rendezvous ports.

Use the buffer port's predefined **get()** and **put()** methods to read or write port values. These methods are TCMs. The **$** port access operator cannot be used with buffer ports.

A typical usage of a buffer port is in a *producer* and *consumer* protocol, where one object puts data on an output port at possibly irregular intervals, and another object with the corresponding input port reads the data at its own rate.

Syntax example:

```
rq : in buffer_port of bool is instance
```

9.6.3 event_port

Purpose	Transfer events between units or between simulators and units	
Category	Unit member	
Syntax	*event-port-field-name* **:** [**list of**] [*direction*] **event_port is instance**	
Parameters	*event-port-field-name*	A unique identifier used to reference the port or access its value.
	direction	One of **in**, **out**, or **inout**. The default is **inout**, which means events can be emitted and sampled on the port. For a port with direction **in**, events can only be sampled. For a port with direction **out**, events can only be emitted.

Event ports can be used to transfer events between two ***e*** units or between an ***e*** unit and an external object. Use the **$** port access operator to read or write port values (see 9.4.1).

An event port’s direction can be either in, out, or inout. Omitting the direction is the same as writing inout. Port types with different directions are not equivalent. The following types are fully equivalent:

```
clk :       event_port is instance;
clk : inout event_port is instance
```

In addition, the following are not allowed:

— Using the **on** struct member for event ports
— Coverage on event ports
— Specifying a temporal formula (e.g., `out event_port is ...`) to define an out event port

It is possible, however, to define an additional event and connect it to the event port, e.g.,

```
ep : in event_port is instance;
  keep bind(ep, external);
event e is @ep$
```

Syntax example:

```
clk : in event_port is instance
```

9.6.4 method_port

Purpose	Enable invocation of abstract functions	
Category	Unit member	
Syntax	*port-instance-name* **:** [**list of**] *direction* **method_port of** *method-type* **is instance**	
Parameters	*port-instance-name*	A unique identifier used to reference the method port or invoke the actual method. For input method ports, this name shall be the same as that of the associated method.
	direction	One of **in** or **out**. There is no default. For an **in** port, only the method to activate can be specified; for an **out** port, the method can be invoked.
	method-type	A *method type* that specifies the port semantics (see also 9.6.5).

Method ports implement an abstraction of the calling methods (time-consuming or not) in other units or external agents, while delaying the binding from compile time to pre-run generation time.

Syntax example:

```
convert_string : out method_port of str2uint_method_t is instance
```

9.6.5 method_type *method-type-name*

Purpose	Associate method prototype with type name and enable notification	
Category	Statement	
Syntax	**method_type** *method-type-name* **(**[*param-list*]**)** [**:***return-type*] [**@sys.any**]	
Parameters	*method-type-name*	A unique identifier used to reference the method type.
	param-list	This needs to match the parameter list of the ***e*** method or external function.
	return-type	This needs to match the return type of the ***e*** method or external function.
	@sys.any	If the method type declaration includes the **@sys.any** sampling event, this method type can only be used for method ports associated with a TCM.

A method port (see 9.6.4) shall be parameterized by a *method type*, which specifies the prototype (signature) of the method. The method type name can also be included in any runtime messages related to a specific method port.

Syntax example:

```
method_type str2uint_method_t(s:string): uint
```

9.6.6 Port reference

Purpose	Reference a port instance	
Category	Unit field, variable, or method parameter	
Syntax	[**!** | **var**] *port-reference-name***:** [*direction*] *port-kind* [**of** *element-type*]	
Parameters	*port-reference-name*	A unique identifier.
	direction	One of **in** or **out**; for simple ports and event ports, this can also be **inout**.
	port-kind	One of **simple_port**, **buffer_port**, or **event_port**.
	element-type	Required if port-kind is **simple_port** or **buffer_port**.

If a port reference is a field, then it shall be marked as non-generated or it needs to be constrained to an existing port instance. Otherwise, a generation error shall result.

Syntax example:

```
!in_int_buffer_port_ref : in buffer_port of int
```

9.6.7 Port: $

Purpose	Read or write a value to a simple port or event port	
Category	Operator	
Syntax	*exp***$**	
Parameters	*exp*	An expression that returns a simple port or event port instance.

The **$** access operator can be used to access or update the value held in a simple port or event port. When used on the RHS, `p$` refers to the port's value. On the LHS of an assignment, `p$` refers to the value's location, so an assignment to `p$` changes the value held in the port.

Without the **$** operator, an expression of any type port refers to the port itself, not to its value. In particular, an expression without the **$** operator can be used for operations involving port references.

Syntax example:

```
p$ = 32'bz          // Assigns an mvl literal to the port 'p'
```

9.6.8 Port bit slice access

Purpose	Read or write a value to a bit or a bit slice of a simple port	
Category	Operator	
Syntax	*simple-port-exp***$[**[*high-exp*]**:**[*low-exp*][**:***slice*]**]**	
Parameters	*simple-port-exp*	An expression that returns a simple port instance. The element type must be scalar, list of bit or list of byte.
	high-exp	A non-negative numeric expression. The high expression must be greater than or equal to *low-exp*. The default value depends on the size of the *exp*. For example, if *exp* is a 32-bit integer and the *slice* is bit, the default value is 32.
	low-exp	A non-negative numeric expression, less than or equal to the high expression. The default value is 0.
	slice	Can be **bit, byte**, **int**, or **uint**. The default is **bit**.

The bit slice operator can be used to extract or update the specified bits of the value held in a simple port.

When the expression appears on the LHS of an assignment, the specified bits in the location that the port refers to are set to the value of the RHS expression. The RHS value is chopped or zero/sign extended, if necessary.

When the expression appears on the RHS, the specified bits in the location that the port refers to are used.

See also 4.12.2, 9.6.7.

Syntax example:

```
print p$[15:0]
```

9.6.9 Force simple port

Purpose	Force a value to a simple port	
Category	Action	
Syntax	**force** *simple-port-exp***$ =** *exp*	
Parameters	*simple-port-exp*	An expression that returns a simple port instance.
	exp	An expression of the same type as the port's type.

This forces the value of *exp* to the port specified by *simple-port-exp*. The value held inside the port is updated immediately. Until the port is released by a **release** action (see 9.6.11), any other non-force assignment to the port will be ignored. Any subsequent force assignment overrides the last one.

Syntax example:

```
force p$ = 0xabc
```

9.6.10 Force simple port bit slice

Purpose	Force a value to a bit or a bit slice of a simple port	
Category	Action	
Syntax	**force** *simple-port-exp***$[**[*high-exp*]**:**[*low-exp*][**:***slice*]**] =** *exp*	
Parameters	*simple-port-exp*	An expression that returns a simple port instance.
	high-exp	A non-negative numeric expression. The high expression must be greater than or equal to *low-exp*. The default value depends on the size of the *exp*. For example, if *exp* is a 32-bit integer and the *slice* is bit, the default value is 32.
	low-exp	A non-negative numeric expression, less than or equal to the high expression. The default value is 0.
	slice	Can be **bit, byte**, **int**, or **uint**. The default is **bit**.
	exp	An expression of the same type as the port's type.

This writes the value of *exp* to the specified bits slice of the port specified by *simple-port-exp*. In addition, it forces the entire port, (similarly to Force simple port – see 9.6.9). That is, until the next release action, any subsequent non-force assignment is ignored. Any subsequent force assignment overrides the last one.

See also 9.6.9, 9.6.8.

Syntax example:

```
force p$[15:0] = 0xabc
```

9.6.11 Release simple port

Purpose	Remove a **force** action from a simple port
Category	Action
Syntax	**release** *simple-port-exp***$**
Parameters	*simple-port-exp* An expression that returns a simple port instance.

This releases the port specified by *simple-port-exp* that previously has been forced (see 9.6.9).

Syntax example:

```
release p$
```

9.6.12 Method port reference

Purpose	Reference a method port instance	
Category	Unit field, variable, or method parameter	
Syntax	[**!** | **var**] *port-reference-name***:** *direction* **method_port of** *method-type*	
Parameters	*port-reference-name*	A unique identifier used to reference the method port.
	direction	One of **in** or **out**.
	method-type	A *method type* that specifies the port semantics (see also 9.6.5).

Method port instances may be referenced by a field, variable, or method parameter of the same port type.

If a port reference is a field, it shall be marked as non-generated, or it needs to be constrained to an existing port instance. Otherwise, a generation error shall result. Also, port binding is allowed only for port instance fields, not for port reference fields (see also 9.5.5).

Syntax example:

```
!in_method_port_ref : in method_port of burst_method_t
```

9.6.13 Method port: $

Purpose	Call an out method port	
Category	Operator	
Syntax	*port-exp*$(*out-method-port-param-list*)	
Parameters	*port-exp*	An expression that returns an output method port instance.
	out-method-port-param-list	A list of actual parameters to the output method port. The number and type of the parameters, if any, shall match the *method type* (see also 9.6.5).

The **$** access operator can be used to call an output method port. An attempt to call a method via the port without using the **$** operator shall result in a syntax error. Without the **$** operator, an expression of any type port refers to the port itself, not to its value. In particular, an expression without the **$** operator can be used for operations involving port references.

Syntax example:

```
    u = convert_string$("32")   //calls the convert_string out method port
```

9.7 Port attributes

Ports have attributes that affect their behavior and how they can be used. Use the *attribute***()** syntax to assign port attributes in pre-generation constraints, as follows:

keep [**soft**] *port_instance*.*attribute*() **==** *value*

Use soft constraints for attributes that can be overridden.

Most port attributes are ignored, unless the port is an external port, but it does no harm to specify attributes for ports that are not external ports. Attributes intended for external ports do not have to be supported for a particular simulator.

9.7.1 Generic port attributes

Port attributes that are potentially valid for all simulators are described in Table 21. However, a particular simulator adapter might not implement some of these attributes. Depending on the simulator adapter, port attributes might cause additional code to be written to the `stubs` file (see Clause 23). In that case, if an attribute is added or changed, the `stubs` file needs to be rewritten.

Table 21—Generic port attributes

Attribute	Description	Applies to
bind()	Connects two internal ports or connect a port to an external object. Type: *bool* Default: none See also 9.7.2.1.	All kinds of internal and external ports
buffer_size()	Specifies the maximum number of elements for a buffer port queue. Type: **uint** Default: none See also 9.7.2.2.	Buffer ports
declared_range()	Specifies the bit width of an external multi-bit object. Type: *string* Default: none See also 9.7.2.3.	External output simple ports that are bound to some kinds of multi-bit objects
delayed()	Specifies whether propagation of a new port value assignment occurs immediately or is delayed to the tick boundary. Type: *bool* Default: `TRUE` See also 9.7.2.4.	Internal and external simple ports
driver()	When `TRUE`, an additional resolved HDL driver is created for the corresponding simulator item, and that driver is written to instead of the port. Type: *bool* Default: `FALSE` See also 9.7.2.5.	External output simple ports
driver_delay()	Specifies the delay time for all assignments from ***e*** to the port. Type: **time** Default: `0` See also 9.7.2.6.	External output simple ports
edge()	Specifies the edge on which an event is generated. Type: **event_port_edge** Default: **change** See also 9.7.2.8.	External input event ports
hdl_convertor()	Specifies the rules for converting method port arguments between ***e*** and a foreign language, such as SystemVerilog or VHDL. The syntax of the string value associated with **hdl_convertor()** is defined by the language adapter itself. Type: *string* Default: none	Method ports
hdl_path()	Specifies a relative path of the corresponding simulated item as a string. Type: *string* Default: none See also 9.7.2.9.	External ports

Table 21—Generic port attributes *(continued)*

Attribute	Description	Applies to
pack_options()	Specifies how the port’s data element is implicitly packed and unpacked. Type: **pack_options** Default: `NULL` See also 9.7.2.10.	External simple ports
pass_by_pointer	When `TRUE`, composite data (structs or lists) are passed by reference. Type: *bool* Default: `FALSE` (pass by value) See also 9.7.2.11.	Internal simple or buffer ports whose data element is a composite type (lists and structs)

9.7.2 Port attributes for HDL simulators

Port attributes that are potentially valid for all HDL simulators are described in Table 22. However, a particular simulator adapter might not implement some of these attributes. The port attributes in Table 22 enable extended functionality. They cause additional information to be written into the HDL `stubs` file to enhance user control over the driving of HDL signals. For this reason, any attribute shown in Table 22 is added or changed, the `stubs` file needs to be rewritten.

Example

The following attributes define a port that is eight bits wide; read operations occur with one-unit delay; drive operations have a five-unit delay:

```
data : inout simple_port of uint(bits:8) is instance;
  keep bind(data, external);
  keep data.hdl_path()       == "sig";
  keep data.declared_range() == "[7:0]";
  keep data.verilog_strobe() == "#1";
  keep data.verilog_drive()  == "#5"
```

Table 22—Port attributes for Verilog or VHDL agents

Attribute	Description	Applies to
driver_initial_value()	Applies an initial mvl value to the port. Type: *list of mvl* Default: **{}** (empty list) See also 9.7.2.7.	External output simple ports
verilog_drive()	Specifies the event on which the data is driven to the Verilog object. Type: *string* Default: none See also 9.7.2.12.	External output simple ports
verilog_drive_hold()	Specifies an event after which the port data is set to `Z`. Type: *string* Default: none See also 9.7.2.13.	External output simple ports

Table 22—Port attributes for Verilog or VHDL agents *(continued)*

Attribute	Description	Applies to
verilog_forcible()	Allows forcing of Verilog wires. Type: *bool* Default: `FALSE` See also 9.7.2.14.	External output simple ports
verilog_strobe()	Specifies the sampling event for the Verilog signal that is bound to the port. Type: *string* Default: none See also 9.7.2.15.	External output simple ports
verilog_wire()	Binds an external out port to a Verilog wire. Type: *bool* Default: `FALSE` See also 9.7.2.16.	External output simple ports
vhdl_delay_mode()	Specifies whether pulses whose period is shorter than the delay are propagated through the driver. Type: **vhdl_delay_mode** Default: **TRANSPORT** (all pulses, regardless of length, are propagated) See also 9.7.2.17.	External output simple ports
vhdl_driver()	This is an alias for the **driver()** attribute. Type: *bool* Default: `FALSE` See also 9.7.2.5.	External output simple ports

9.7.2.1 bind()

Purpose	Connect two internal ports or connect a port to an external object	
Category	Generic port attribute	

Syntax	**bind(***exp1***,** *exp2***)**
	bind(*exp1***,** (**external** | **empty** | **undefined**)**)**	

Parameters	*exp1***,** *exp2*	One or more expressions of port type. If two expressions are given and the port types are compatible, the two port instances are connected.
	external	Defines a port as connected to a simulated object, such as a Verilog register, VHDL signal, or SystemC object.
	empty	Defines a disconnected port. Runtime accessing of a port with an empty binding is allowed.
	undefined	Defines a disconnected port. Runtime accessing of a port with an undefined binding shall cause an error.

Ports are connected to other ***e*** ports or to external simulated objects, such as Verilog registers, VHDL signals, or SystemC methods, using a pre-run generation constraint on the **bind()** attribute. Ports can also be left explicitly disconnected by using **empty** or **undefined**.

9.7.2.1.1 Rules

a) All ports shall be bound in one of the following ways:
   1) A set of bound ports must include at least two ports, one of which is an input or inout port, and the other an output or inout port.
   2) Bound to an external simulated item.
   3) Explicitly disconnected (empty or undefined).
b) Only ports of the same kind can be bound together. A simple port cannot be bound to a buffer port or an event port, and a buffer port cannot be bound to an event port.
c) Dangling ports [ports without **bind()** attributes] shall cause an error during elaboration (see 9.7.2.1.2).
d) A port can be explicitly disconnected and then overridden with a binding to an internal or external object.
e) All ports connected together shall have the exact same element type.

9.7.2.1.2 Checking of ports

Binding and checking of ports takes place automatically at the end of the predefined **generate_test()** test method. This process, called *elaboration of ports*, includes checking for dangling ports and binding consistency (directions, buffer sizes, and so on).

A port that has no **bind()** constraint is a *dangling port*. Since all ports need to be bound, a dangling port shall cause an elaboration-time error.

9.7.2.1.3 Disconnected ports

A port that is bound using the **empty** or **undefined** keyword is called a *disconnected port*. The **empty** or **undefined** keyword can only appear as the second argument of the **bind()** constraint, in place of a second port instance name.

Empty binding can be used to define a port that is connected to nothing. Runtime accessing of an empty-bound port is allowed. Its effect depends on the operation and type of the port.

— Reading from an empty-bound simple port returns the last written value or the default of the port element type, if no value has been written so far.
— Writing to an empty-bound output or inout simple port stores the new value internally.
— Reading from an empty-bound buffer port causes the thread to halt.
— Writing to an empty-bound buffer port causes the thread to halt if the buffer is full.
— Waiting for an empty-bound event port causes the thread to halt. If the port direction is inout, then emitting the port resumes the thread.
— An empty-bound event port can be emitted.

A subsequent constraint can be used to overwrite the empty binding constraint.

Like empty binding, undefined binding can define a port that is connected to nothing. The difference is runtime accessing of a port with an undefined binding shall cause an error.

A subsequent constraint can be used to overwrite the undefined binding constraint.

Syntax example:

```
buf_in1 : in buffer_port of int(bits:16) is instance;
```

```
    keep bind(buf_in1, empty)
```

9.7.2.2 buffer_size()

Purpose	Specify the size of a buffer port queue	
Category	Buffer port attribute	
Syntax	*exp*.**buffer_size() ==** *num*	
Parameters	*exp*	An expression of type [**in** | **out**] **buffer_port of** *type*.
	num	An integer specifying the maximum number of elements for the queue.

This attribute determines the number of **put()** actions that can be performed before a **get()**. A **get()** action is required to remove data and make more room in the queue. Specifying a buffer size of `0` means rendezvous-style synchronization.

No default buffer size is provided. If a buffer size is not specified in a constraint, an error shall occur. It is only necessary to specify a buffer size for one of the two ports in a pair of connected ports. That size applies to both ports. If the two ports have different buffer sizes specified, then both of them get the larger of the two sizes.

Syntax example:

```
  keep u.p.buffer_size() == 20
```

9.7.2.3 declared_range()

Purpose	Specify the bit width of a multi-bit external object	
Category	External port attribute	
Syntax	*exp*.**declared_range() ==** *string*	
Parameters	*exp*	An expression of a simple port type.
	string	A string that is a valid range expression, e.g., `"[`*msb*`:`*lsb*`]"`

This string attribute is meaningful for external simple ports that are bound to multi-bit objects. Because it is legal to bind a port to an HDL object with a different size, the range information is not extracted from the port declaration. In order to implement access to multi-bit signals correctly in the `stubs` file (see Clause 23), this attribute is required when using the **verilog_wire()**, **verilog_drive()**, **verilog_strobe()**, or **driver()** attributes.

The interpretation of the string is simulator-specific.

Syntax example:

```
  keep u.p.declared_range() == "[31:0]"
```

9.7.2.4 delayed()

Purpose	Specify immediate or delayed propagation of new values
Category	Simple port attribute
Syntax	*exp*.**delayed()** **not** *exp*.**delayed()** *exp*.**delayed() ==** *bool*
Parameters	*exp* — An expression of a simple port type.
	bool — Either `TRUE` or `FALSE`. The default is `TRUE`.

This Boolean attribute specifies whether propagation of a new port value assignment occurs immediately or is delayed. When the **delayed()** attribute is `TRUE` (the default), propagation of external ports is delayed until the next tick. Propagation of internal ports is delayed until the next tick when the **sys.time** value changes. This behavior is consistent with the definition of delayed assignments in ***e*** and matches temporal ***e*** semantics with regard to the multiple ticks occurring at the same simulator time.

To make assigned values on ports visible immediately, constrain this attribute to be `FALSE`.

Syntax example:

```
keep not u.p.delayed()
```

9.7.2.5 driver()

Purpose	Create a resolved driver for an external object
Category	External out port attribute
Syntax	*exp*.**driver()** **not** *exp*.**driver()** *exp*.**driver() ==** *bool*
Parameters	*exp* — An expression of a simple port type.
	bool — Either `TRUE` or `FALSE`. The default is `FALSE`.

This Boolean attribute is meaningful only for external out ports. When this attribute is set to `TRUE`, an additional resolved HDL driver is created for the corresponding simulator item and that driver is written to instead of the port.

Every port instance associated with the same simulator can create a separate driver, thus allowing HDL resolution to be applied for multiple ***e*** resources.

Syntax example:

```
keep u.p.driver()
```

9.7.2.6 driver_delay()

Purpose	Specify the delay for assignments to a port	
Category	External out simple port attribute	
Syntax	*exp*.**driver_delay() ==** *time*	
Parameters	*exp*	An expression of a simple port type.
	time	A value of type **time** (64 bits). The default is `0`.

This attribute is meaningful only for external out ports. It specifies the delay time for all assignments from ***e*** to the port. This attribute is silently ignored, unless the **driver()** attribute or the **vhdl_driver()** attribute is set to `TRUE`.

Syntax example:

```
    keep u.p.driver_delay() == 2
```

9.7.2.7 driver_initial_value()

Purpose	Specify an initial value for an HDL object	
Category	HDL port attribute	
Syntax	*exp*.**driver_initial_value() ==** *mvl-list*	
Parameters	*exp*	An expression that returns a port instance.
	mvl-list	A lists of mvl values. The default is **{}** (an empty list).

This *mvl-list* type attribute applies an initial mvl value to an external Verilog or VHDL object. This attribute is silently ignored, unless the **driver()** attribute or the **vhdl_driver()** attribute is set to `TRUE`.

The default value of this attribute is `MVL_X`.

Syntax example:

```
    keep u.p.driver_initial_value() == {MVL_X; MVL_X; MVL_1; MVL_1}
```

9.7.2.8 edge()

Purpose	Specify the edge on which an event is generated	
Category	Event port attribute	
Syntax	*exp*.**edge() ==** *edge-option*	
Parameters	*exp*	An expression of a **buffer_port** type.
	edge-option	A value of type **event_port_edge**.

This attribute of type **event_port_edge** (for an external event port) specifies the edge on which an event is generated. The possible values are as follows:

a) **change**, **rise**, **fall**—equivalent to the behavior of **@sim** TEs. This means that transitions between `x` and `0`, `z`, and `1` are not detected; `x` to `1` is considered a rise; `z` to `0` a fall, and so on.
b) **any_change**—any change within the supported MVL values is detected, including transitions from `x` to `0` and `1` to `z`.
c) **MVL_0_to_1**—transitions from `0` to `1` only.
d) **MVL_1_to_0**—transitions from `1` to `0` only.
e) **MVL_X_to_0**—transitions from `X` to `0` only.
f) **MVL_0_to_X**—transitions from `0` to `X` only.
g) **MVL_Z_to_1**—transitions from `Z` to `1` only.
h) **MVL_1_to_Z**—transitions from `1` to `Z` only.

The default is **change**.

Syntax example:

```
keep e.edge() == any_change
```

9.7.2.9 hdl_path()

Purpose	Map port instance to an external object	
Category	Generic port attribute	
Syntax	*exp*.**hdl_path() ==** *string*	
Parameters	*exp*	An expression of a port type.
	string	A string specifying the path to the external object. The default is an empty string.

This attribute specifies a path for accessing an external, simulated object. The path is a concatenation of the partial paths for the port itself and its enclosing units. The partial paths can use any supported separator. To allow portability between simulators, use the ***e*** canonical path notation.

Syntax example:

```
clk : in event_port is instance;
  keep clk.hdl_path() == "clk"
```

9.7.2.10 pack_options()

Purpose	Specify how an external port's data element is implicitly packed and unpacked	
Category	External simple port attribute	
Syntax	*exp*.**pack_options() ==** *pack-option*	
Parameters	*exp*	An expression of a simple or buffer port type.
	pack-option	A predefined or user-defined pack option. The default is `NULL`.

This attribute can be used to specify the way that data element of external ports is implicitly packed and unpacked. This attribute exists both for units and ports, and can be propagated downwards from an enclosing unit instance to its ports and other unit instances.

Syntax example:

```
keep u.p.pack_options() == packing.low_big_endian
```

9.7.2.11 pass_by_pointer()

Purpose	Specify how composite data is transferred by internal ports	
Category	Internal port attribute	
Syntax	*exp*.**pass_by_pointer()** **not** *exp*.**pass_by_pointer()** *exp*.**pass_by_pointer() ==** *bool*	
Parameters	*exp*	An expression of a simple or buffer port type.

This Boolean attribute specifies how composite data (structs or lists) is transferred by internal simple ports or buffer ports. By default, this attribute is `FALSE` and complex objects are deep-copied upon an internal port access operation. To pass data by reference and speed up the test, set this attribute to `TRUE` (and verify no test-correctness violations exist).

Syntax example:

```
keep u.p.pass_by_pointer();
keep not u.p.pass_by_pointer()
```

9.7.2.12 verilog_drive()

Purpose	Specify timing control for data driven to a Verilog object	
Category	Verilog port attribute	
Syntax	*exp*.**verilog_drive() ==** *timing-control*	
Parameters	*exp*	An expression of a simple port type.
	timing-control	A string specifying any legal Verilog timing control (event or delay).

This string attribute tells an external output port to drive its data to a Verilog signal when the specified timing occurs. This can be a Verilog TE, such as `@(posedge top.clk)`, or a simple unit delay, e.g., `#1`.

Syntax example:

```
keep u.p.verilog_drive() == "@posedge clk2"
```

9.7.2.13 verilog_drive_hold()

Purpose	Specify when to set the port to Z	
Category	Verilog port attribute	
Syntax	*exp*.**verilog_drive_hold() ==** *string*	
Parameters	*exp*	An expression of a simple port type.
	string	A string specifying any legal Verilog timing control.

On the first occurrence of the specified event after the port data is driven, the value of the corresponding Verilog signal is set to `Z`. The event is a string specifying any legal Verilog timing control. The **verilog_drive()** attribute (see 9.7.2.12) needs to be specified before using this attribute.

Syntax example:

```
keep u.p.verilog_drive_hold() == "@negedge clk2"
```

9.7.2.14 verilog_forcible()

Purpose	Specify a Verilog object can be forced	
Category	Verilog port attribute	
Syntax	*exp*.**verilog_forcible()** **not** *exp*.**verilog_forcible()** *exp*.**verilog_forcible() ==** *bool*	
Parameters	*exp*	An expression of a simple port type.
	bool	Either `TRUE` or `FALSE`. The default is `FALSE`.

By default, Verilog wires are not forcible. This Boolean attribute allows forcing of Verilog wires. The **verilog_wire()** attribute (see 9.7.2.16) needs to be specified before using this attribute.

Syntax example:

```
keep u.p.verilog_forcible()
```

9.7.2.15 verilog_strobe()

Purpose	Specify the sampling event for a Verilog object	
Category	Verilog port attribute	
Syntax	*exp*.**verilog_strobe() ==** *string*	
Parameters	*exp*	An expression of a simple port type.
	string	A string specifying any legal Verilog timing control.

This string attribute specifies the sampling event for the Verilog signal that is bound to an external input port. This attribute is equivalent to the **verilog variable ... using strobe** declaration.

Syntax example:

```
keep u.p.verilog_strobe() == "@posedge clk1"
```

9.7.2.16 verilog_wire()

Purpose	Create a single driver for a port (or multiple ports)	
Category	Verilog port attribute	
Syntax	*exp*.**verilog_wire()** **not** *exp*.**verilog_wire()** *exp*.**verilog_wire() ==** *bool*	
Parameters	*exp*	An expression of a simple port type.
	bool	Either `TRUE` or `FALSE`. The default is `FALSE`.

This Boolean attribute allows an external out port to be bound to a Verilog wire. The main difference between this attribute and the **driver()** attribute is the **verilog_wire()** attribute merges all of the ports containing this attribute into a single Verilog driver, while the **driver()** attribute creates a separate driver for each port.

Syntax example:

```
keep u.p.verilog_wire()
```

9.7.2.17 vhdl_delay_mode()

Purpose	Specify whether short pulses are propagated through the driver	
Category	HDL port attribute	
Syntax	*exp*.**vhdl_delay_mode() ==** *mode-option*	
Parameters	*exp*	An expression of a simple port type.
	mode-option	Either **TRANSPORT** (the default) or **INERTIAL**.

This attribute specifies whether pulses whose period is shorter than the delay specified by the **driver_delay()** attribute are propagated through the driver. **INERTIAL** specifies such pulses are not propagated, **TRANSPORT** that all pulses, regardless of length, are propagated.

This attribute also influences what happens if another driver (either VHDL or another unit) schedules a signal change, and before that change occurs, this driver schedules a different change. With **INERTIAL**, the first change never occurs.

This attribute is silently ignored, unless the **driver_delay()** attribute is also specified.

Syntax example:

```
keep u.p.vhdl_delay_mode() == INERTIAL
```

9.8 Buffer port methods

The following methods are used to read from or write to buffer ports, and to check whether a buffer port queue is empty or full.

9.8.1 get()

Purpose	Read and remove data from an input buffer port queue	
Category	Predefined TCM for buffer ports	
Syntax	*in-port-instance-name*.**get()**: port element type	
Parameters	*in-port-instance-name*	An expression that returns an input buffer port instance.

Reads a data item from the buffer port queue and removes the item from the queue. Since buffer ports use a FIFO queue, **get()** returns the first item that was written to the port.

The thread blocks upon **get()** when there are no more items in the queue. If the queue is empty, or if it has a buffer size of `0` and no **put()** has been done on the port since the last **get()**, then the **get()** is blocked until a **put()** is done on the port.

The number of consecutive **get()** actions that is possible is limited to the number of items inserted by **put()**.

Syntax example:

```
    rec_cell = in_port.get()
```

9.8.2 put()

Purpose	Write data to an output buffer port queue	
Category	Predefined TCM for buffer ports	
Syntax	*out-port-instance-name*.**put()**(*data*: port element type)	
Parameters	*out-port-instance-name*	An expression that returns an output buffer port instance.
	data	A data item of the port element type.

Writes a data item to the output buffer port queue. The sampling event of this TCM is **sys.any**. The new data item is placed in a FIFO queue in the output buffer port.

The thread blocks upon **put()** when there is no more room in the queue, i.e., when the number of consequent **put()** operations exceeds the **buffer_size()** of the port instance. If the queue is full, or if it has a buffer size of `0` and no **get()** has been done on the port since the last **put()**, then the **put()** is blocked until a **get()** is done on the port.

The number of consecutive **put()** actions that is possible is limited to the buffer size.

Syntax example:

```
    out_port.put(trans_cell)
```

9.8.3 is_empty()

Purpose	Check if an output buffer port queue is empty
Category	Pseudo-method for buffer ports
Syntax	*in-port-instance-name*.**is_empty()** **not** *in-port-instance-name*.**is_empty()** *in-port-instance-name*.**is_empty() ==** *bool*
Parameters	*in-port-instance-name* An expression that returns an input buffer port instance.

Returns TRUE if the input port queue is empty. Returns FALSE if the input port queue is not empty.

Syntax example:

```
var readable : bool;
readable = not cell_in.is_empty()
```

9.8.4 is_full()

Purpose	Check if an output buffer port queue is full
Category	Pseudo-method for buffer ports
Syntax	*out-port-instance-name*.**is_full()** **not** *out-port-instance-name*.**is_full()** *out-port-instance-name*.**is_full() ==** *bool*
Parameters	*out-port-instance-name* An expression that returns an output buffer port instance.

Returns TRUE if the output port queue is full. Returns FALSE if the output port queue is not full.

Syntax example:

```
var overflow : bool;
overflow = cell_out.is_full()
```

9.9 MVL methods for simple ports

The predefined port methods in this subclause are for reading and writing MVL data between ports, to facilitate communication with objects where MVL values occur. These methods operate on data of type mvl, which is defined as follows:

```
type mvl : [MVL_U, MVL_X, MVL_0, MVL_1, MVL_Z, MVL_W, MVL_L, MVL_H, MVL_N]
```

The enumeration literals are the same as those of VHDL, except for MVL_N, which corresponds to the VHDL-(“don’t care”) literal.

The MVL methods are applicable according to the port direction. Methods that write a value to a port are accessible for output and inout simple ports, while methods that read a value from a port are accessible for input and inout simple ports.

Accessing a port with MVL methods and accessing it through the **$** operator is allowed (*mixed access*).

9.9.1 MVL four-value logic

Some MVL methods operate on a subset of the enumeration in 9.9, `MVL_X`, `MVL_Z`, `MVL_0`, and `MVL_1`, which corresponds to the four-value logic of Verilog. To convert from nine-value logic to four-value logic, the mapping shown in Table 23 is used.

Table 23—MVL logic mapping

Nine value	Four value
`MVL_U`, `MVL_W`, `MVL_X`, `MVL_N`	`MVL_X`
`MVL_L`, `MVL_0`	`MVL_0`
`MVL_H`, `MVL_1`	`MVL_1`
`MVL_Z`	`MVL_Z`

9.9.2 MVL string

Several functions allow specifying the MVL value or returning an MVL value expressed as string. A format of MVL string is the number of bits followed by the **'** sign, the radix, and then the MVL literals. When an MVL list is converted into a string, the mapping shown in Table 24 is used.

The mapping is done in the following way:

Table 24—MVL string conversion

MVL value	String
`MVL_U`	`u`
`MVL_X`	`x`
`MVL_0`	`0`
`MVL_1`	`1`
`MVL_Z`	`z`
`MVL_W`	`w`
`MVL_L`	`L`
`MVL_H`	`h`
`MVL_N`	`n`

When a string is converted to a list of mvl, the mapping is case-insensitive.

9.9.3 put_mvl()

Purpose	Put an mvl data on a port of a non-mvl type	
Category	Predefined method for simple ports	
Syntax	*exp*.**put_mvl(***value*: mvl**)**	
Parameters	*exp*	An expression that returns a simple port instance.
	value	An mvl value.

Places an mvl value on an output or inout simple port, e.g., to initialize an object to a “disconnected” value. Placing an mvl value on a port whose element type is wider than one bit places the value in the LSB of the element.

Syntax example:

```
p.put_mvl(MVL_Z)
```

9.9.4 get_mvl()

Purpose	Read mvl data from a port of a non-mvl type	
Category	Predefined method for simple ports	
Syntax	*exp*.**get_mvl()**: mvl	
Parameters	*exp*	An expression that returns a simple port instance.

Reads an mvl value from an input or inout simple port, e.g., to check that there are no undefined `x` bits. Getting an mvl value on a port whose element type is wider than one bit returns the value in the LSB of the element.

Syntax example:

```
check that pbi.get_mvl() != MVL_X else dut_error("Bad value")
```

9.9.5 put_mvl_list()

Purpose	Put a list of mvl values on a port of a non-mvl type	
Category	Predefined method for simple ports	
Syntax	*exp*.**put_mvl_list(***values*: list of mvl**)**	
Parameters	*exp*	An expression that returns a simple port instance.
	values	A list of mvl values.

Writes a list of mvl values to an output or inout simple port. Putting a list of mvl values on a port whose element type is a single bit writes only the LSB of the list.

Syntax example:

```
    pbo.put_mvl_list({MVL_H; MVL_0; MVL_L; MVL_0})
```

9.9.6 fill_mvl_list()

Purpose	Get a list of mvl values from a port of a non-mvl type, and fill an existing buffer list with those values.	
Category	Predefined method for simple ports	
Syntax	*exp***.fill_mvl_list(***mvl-list*: list of mvl**)**	
Parameters	*exp*	An expression that returns a simple port instance.
	mvl-list	A list of mvl values.

Reads a list of mvl values from an input or inout simple port and puts the values into *mvl-list*. If a non-empty list is passed, its initial content is cleared and disregarded.

Syntax example:

```
var mvll: list of MVL;
pbil.fill_mvl_list(mvll)
```

9.9.7 get_mvl_list()

Purpose	Get a list of mvl values from a port of a non-mvl type	
Category	Predefined method for simple ports	
Syntax	*exp*.**get_mvl_list()**: list of mvl	
Parameters	*exp*	An expression that returns a simple port instance.

Reads a list of mvl values from an input or inout simple port.

Syntax example:

```
check that not pbil.get_mvl_list().has(it == MVL_U)
  else dut_error("Bad list")
```

9.9.8 put_mvl_string()

Purpose	Put an mvl value on a port of a non-mvl type when a value is represented as a string	
Category	Predefined method for simple ports	
Syntax	*exp*.**put_mvl_string(***value*: string**)**	
Parameters	*exp*	An expression that returns a simple port instance.
	value	An mvl value in the form of a base and one or more characters, entered as a string. The mvl values in the string shall be lowercase. Use `1` for `MVL_1`, `0` for `MVL_0`, `z` for `MVL_Z`, and so on.

Writes a string representing a list of mvl values to a simple output or inout port. See also 9.9.2.

Syntax example:

```
pbol.put_mvl_string("32'hxxxxllll")
```

9.9.9 get_mvl_string()

Purpose	Get a value in form of a string from a port of a non-mvl type	
Category	Predefined method for simple ports	
Syntax	*exp*.**get_mvl_string(***radix*: radix**)**: string	
Parameters	*exp*	An expression that returns a simple port instance.
	radix	One of `BIN`, `OCT`, or `HEX`.

Returns a string in which each character represents an mvl value. See also 9.9.2.

Syntax example:

```
print pbis.get_mvl_string(BIN)
```

9.9.10 get_mvl4()

Purpose	Get an mvl value from a port, converting nine-value logic to four-value logic	
Category	Predefined method for simple ports	
Syntax	*exp*.**get_mvl4()**: mvl	
Parameters	*exp*	An expression that returns a simple port instance.

Reads a nine-value mvl value from an input simple port and converts it to four-value subset mvl. See also 9.9.1.

Syntax example:

```
check that pbi.get_mvl4() != MVL_Z else dut_error("Bad value")
```

9.9.11 fill_mvl4_list()

Purpose	Get a list of mvl values from a port, converting it from 9-value logic to 4-value logic, and fill an existing buffer list with those values.	
Category	Predefined method for simple ports	
Syntax	*exp*.**fill_mvl4_list(***mvl-list*: list of mvl**)**	
Parameters	*exp*	An expression that returns a simple port instance.
	mvl-list	A list of mvl valuesto be filled.

Reads a list of 9-value mvl values from an input simple port, converts them to 4-value logic, and puts the values into mvl-list. If a non-empty list is passed, its initial content is cleared and disregarded. See also 9.9.1.

Syntax example:

```
var mvll: list of MVL;
pbi41.fill_mvl4_list(mvll)
```

9.9.12 get_mvl4_list()

Purpose	Get a list of mvl value from a port, converting nine-value logic to four-value logic	
Category	Predefined method for simple ports	
Syntax	*exp*.**get_mvl4()**: list of mvl	
Parameters	*exp*	An expression that returns a simple port instance.

Reads a list of nine-value mvl values from an input simple port and converts them to four-value mvl. See also 9.9.1.

Syntax example:

```
check that not pbi4l.get_mvl4_list().has(it == MVL_X)
  else dut_error("Bad list")
```

9.9.13 get_mvl4_string()

Purpose	Get a four-state value in form of a string from a port of a non-mvl type	
Category	Predefined method for simple ports	
Syntax	*exp*.**get_mvl4_string(***radix*: radix**)**: string	
Parameters	*exp*	An expression that returns a simple port instance.
	radix	One of `BIN`, `OCT`, or `HEX`.

Returns a string representing a four-value logic value. The mvl are first converted into four-value logic (see 9.9.1) and then converted to a string (see 9.9.2).

The returned string always includes all the bits, with no implicit extensions. For example, a port of type `int` returns a string of 32 characters, since an `int` is a 32-bit data type.

Syntax example:

```
print pbi4s.get_mvl4_string(BIN)
```

9.9.14 put_mvl_to_bit_slice()

Purpose	Write mvl data to a bit slice of a port	
Category	Predefined method for simple ports	
Syntax	*exp*.**put_mvl_to_bit_slice(***high*: int, *low*: int, *value*: list of mvl**)**	
Parameters	*exp*	An expression that returns a simple port instance. The element type has to be scalar, list of bit, or list of byte.
	high	An integer that specifies the high index of the bit slice.
	low	An integer that specifies the low index of the bit slice.
	value	list of mvl.

This writes an mvl list to a specified bit slice of the port.

If the size of *value* is smaller than the slice size (1 + *high* - *low*), the value is padded with MVL_Us. If the size of the value is larger than the slice size, then the MSBs of value are truncated.

See also 9.6.8

Syntax example:

```
p.put_mvl_to_bit_slice(7,0,8'hxx)
```

9.9.15 force_mvl()

Purpose	Force mvl data on a port of a non-mvl type	
Category	Predefined method for simple ports	
Syntax	*exp*.**force_mvl(***value*: mvl**)**	
Parameters	*exp*	An expression that returns a simple port instance.
	value	An mvl value.

Forces an mvl value to the specified simple port. The value held inside the port is updated immediately. Until the port is released by a **release** action (see 9.6.11), any other non-force assignment to the port is ignored. Any subsequent force assignment (see 9.6.9, 9.6.10) overrides the last one.

Syntax example:

```
p.force_mvl(MVL_Z)
```

9.9.16 force_mvl_list()

Purpose	Force a list of mvl data on a port of a non-mvl type	
Category	Predefined method for simple ports	
Syntax	*exp*.**force_mvl_list(***values*: list of mvl**)**	
Parameters	*exp*	An expression that returns a simple port instance.
	values	A list of mvl values.

Forces a list of mvl values to the specified simple port. The value held inside the port is updated immediately. Until the port is released by a **release** action (see 9.6.11), any other non-force assignment to the port is ignored. Any subsequent force assignment (see 9.6.9, 9.6.10) overrides the last one.

Syntax example:

```
pbo.force_mvl_list({MVL_H; MVL_0; MVL_L; MVL_0})
```

9.9.17 force_mvl_string()

Purpose	Force an mvl value on a port of a non-mvl type when the value is represented as a string	
Category	Predefined method for simple ports	
Syntax	*exp*.**force_mvl_string(***value*: string**)**	
Parameters	*exp*	An expression that returns a simple port instance.
	value	An mvl value in the form of a base and one or more characters, entered as a string. The mvl values in the string are lowercase. Use 1 for MVL_1, 0 for MVL_0, z for MVL_Z, and so on

Forces a string representing a list of mvl values to the specified simple port. For both e2e and external ports, updates the value held inside the port. The value held inside the port is updated immediately. Until the port is released by a **release** action (see 9.6.11), any other non-force assignment to the port is ignored. Any subsequent force assignment (see 9.6.9, 9.6.10) overrides the last one.

Syntax example:

```
pbol.force_mvl_string("32'hxxxxllll")
```

9.9.18 force_mvl_to_bit_slice()

Purpose	Forces mvl data to a bit slice of a port	
Category	Predefined method for simple ports	
Syntax	*exp*.**force_mvl_to_bit_slice(***high*: int, *low*: int, *value*: list of mvl**)**	
Parameters	*exp*	An expression that returns a simple port instance. The element type must be scalar, list of bit or list of byte.
	high	An integer that specifies the high index of the bit slice.
	low	An integer that specifies the low index of the bit slice.
	value	A list of mvl.

This forces an mvl list to the specified bits slice of the port. The value held inside the port is updated immediately. Until the port is released by a **release** action (see 9.6.11), any other non-force assignment to the port is ignored. Any subsequent force assignment overrides the last one. See also 9.6.10.

Syntax example:

```
p.force_mvl_to_bit_slice(7,0,8'hxx)
```

9.9.19 has_mvl_value()

Purpose	Determine if port has a given MVL value	
Category	Predefined method for simple ports	
Syntax	*exp*.**has_mvl_value(***value*: mvl**)**: bool	
Parameters	*exp*	An expression that returns a simple port instance.
	value	An mvl value.

Returns TRUE if at least one bit of the port equals the given value.

Syntax example:

```
print pbi4s.has_mvl_value(MVL_Z);
```

9.9.20 has_x()

Purpose	Determine if a port has X
Category	Predefined method for simple ports
Syntax	*exp*.**has_x()**: bool
Parameters	*exp* An expression of a simple port type.

Returns `TRUE` if at least one bit of the port is `MVL_X`.

Syntax example:

```
print pbi4s.has_x()
```

9.9.21 has_z()

Purpose	Determine if a port has Z
Category	Predefined method for simple ports
Syntax	*exp*.**has_z()**: bool
Parameters	*exp* An expression of a simple port type.

Returns `TRUE` if at least one bit of the port is `MVL_Z`.

Syntax example:

```
print pbi4s.has_z()
```

9.9.22 has_unknown()

Purpose	Determine if a port has an unknown value
Category	Predefined method for simple ports
Syntax	*exp*.**has_unknown()**: bool
Parameters	*exp* An expression of a simple port type.

Returns `TRUE` if at least one bit of the port is one of the following: `MVL_U`, `MVL_X`, `MVL_Z`, `MVL_W`, or `MVL_N`.

Syntax example:

```
print pbi4s.has_unknown()
```

9.9.23 set_default_value()

Purpose	Set the default value for a simple port or a bound set of simple port	
Category	Predefined method for simple ports	
Syntax	*exp*.**set_default_value(***value*: port-element-type**)**	
Parameters	*exp*	An expression that returns a simple port instance.
	value	An expression of the same type as the port element.

This sets the default value of a simple port or a bound set of simple ports. The default value is stored as the value of the port or ports during the **connect_ports()** phase before simulation begins. If **set_default_value()** is not called, the default value is the same as the default value of the element type (usually 0 or NULL).

The default value applies if an input port is read before it is written. The typical use of these methods is to set the value read from a disconnected port.

When **set_default_value()** is applied to a port in a bound set, the default value is applied to the entire bound set.

If **set_default_value**() is applied to a bound set several times, then each new operation overrides the previous ones.

When two ports that belong to different bound sets are bound to form a new bound set:

— If both ports have default values defined and those values are different, the default value of the new bound set is considered invalid. The default value must be set again or an elaboration time error occurs
— If both ports have the same default value, it becomes the default value of the newly bound set.

  Note: In the above two cases, if one default value was set using **set_default_mvl_value()**, and another one was set using **set_default_value()**, the default values are always considered different. The default value has to be set again after the binding to avoid elaboration time error.

— If only one port has a default value, then this becomes the default value of the new bound set.
— If no port has a default value, then the new bound set has no default value.

If **set_default_value()** is called at any time other than during **connect_ports()** phase, a run-time error shall be issued.

Syntax example:

```
p.set_default_value(15)
```

9.9.24 set_default_mvl_value()

Purpose	Set the default value for a simple port or a bound set of simple port	
Category	Predefined method for simple ports	
Syntax	*exp*.**set_default_mvl_value(***mvl_list*: list of mvl**)**	
Parameters	*exp*	An expression that returns a simple port instance.
	mvl_list	An expression of type list of mvl.

This sets the default value of a simple port or a bound set of simple ports. The default value is stored as the value of the port or ports during the **connect_ports()** phase before simulation begins. If **set_default_value()** is not called, the default value is the same as the default value of the element type (usually 0 or NULL).

The default value applies if an input port is read before it is written. The typical use of these methods is to set the value read from a disconnected port. (See 9.7.2.1.3)

When **set_default_mvl_value()** is applied to a port in a bound set, the default value is applied to the entire bound set.

If **set_default_mvl_value(**) is applied to a bound set several times, then each new operation overrides the previous ones.

If **set_default_value()** is called at any time other than during **connect_ports()** phase, a run-time error shall be issued.

Syntax example:

```
    p.set_default_mvl_value(32'huuuuxxzz)
```

9.10 Global MVL routines

The subclause describes the global routines for manipulating MVL values.

9.10.1 string_to_mvl()

Purpose	Convert a string to a list of mvl values	
Category	Predefined routine	
Syntax	**string_to_mvl(***value-string*: string**)**: list of mvl	
Parameters	*value-string*	A string representing mvl values.

Converts a string into a list of mvl (see 9.9.2).

Syntax example:

```
    mlist = string_to_mvl("8'bxz1")
```

9.10.2 mvl_to_string()

Purpose	Convert a list of mvl values to a string	
Category	Predefined routine	
Syntax	**mvl_to_string(***mvl-list*: list of mvl, *radix*: radix**)**: string	
Parameters	*mvl-list*	A list of mvl values.
	radix	One of `BIN`, `OCT`, or `HEX`.

Converts a list of mvl values to a string (see 9.9.2). A sized number shall always be returned as a string.

Syntax example:

```
mstring = mvl_to_string({MVL_Z; MVL_Z; MVL_Z; MVL_Z;
                         MVL_X; MVL_X; MVL_X; MVL_X}, BIN)
```

9.10.3 mvl_to_int()

Purpose	Convert a list of mvl to an integer	
Category	Predefined routine	
Syntax	**mvl_to_int(***mvl-list*: list of mvl, *mask*: list of mvl**)**: uint	
Parameters	*mvl-list*	A list of mvl values to convert to an integer value.
	mask	A list of mvl values that are to be converted to `1`.

Converts each value in a list of mvl values into a bit (`1` or `0`), using a list of mvl mask values to determine which mvl values are converted to `1`.

When the list is less than 32 bits, it is padded with `0`’s. When it is greater than 32 bits, it is truncated, leaving the 32 least-significant bits.

Syntax example:

```
var ma : uint = mvl_to_int(l, {MVL_X})
```

9.10.4 int_to_mvl()

Purpose	Convert an integer value to a list of mvl values	
Category	Predefined routine	
Syntax	**int_to_mvl(***value*: uint, *mask*: mvl**)**: list of mvl	
Parameters	*value*	An integer value to convert to a list of mvl values.
	mask	An mvl value that replaces each bit in the integer that has the value `1`.

Maps each bit that has the value `1` to the mask mvl value, retains the `0` bits as `MVL_0`, and returns a list of 32 mvl values. The returned list always has a size of 32.

Syntax example:

```
var mlist : list of mvl = int_to_mvl(12, MVL_X)
```

9.10.5 mvl_to_bits()

Purpose	Convert a list of mvl values to a list of bits	
Category	Predefined routine	
Syntax	**mvl_to_bits(***mvl-list*: list of mvl, *mask*: list of mvl**)**: list of bit	
Parameters	*mvl-list*	A list of mvl values to convert to bits.
	mask	A list of mvl values that specifies which mvl values are to be converted to `1`.

Converts a list of mvl values to a list of bits, using a mask of mvl values to indicate which mvl values are converted to `1` in the list of bits.

Syntax example:

```
var bl : list of bit = mvl_to_bits({MVL_Z; MVL_Z; MVL_X; MVL_L},
                                   {MVL_Z; MVL_X})
```

9.10.6 bits_to_mvl()

Purpose	Convert a list of bits to a list of mvl values	
Category	Predefined routine	
Syntax	**bits_to_mvl(***bit-list*: list of bit, *mask*: mvl**)**: list of mvl	
Parameters	*bit-list*	A list of bits to convert to mvl values.
	mask	An mvl value that replaces each bit in the list that has the value `1`.

Maps each bit with the value `1` to the mask mvl value, retains the `0` bits as `MVL_0`, and returns an mvl list that has a size of *bit-list*.

Syntax example:

```
var ml : list of mvl = bits_to_mvl({1; 0; 1; 0}, MVL_Z)
```

9.10.7 mvl_to_mvl4()

Purpose	Convert an mvl value to a four-value logic value	
Category	Predefined routine	
Syntax	**mvl_to_mvl4(***value*: mvl**)**: mvl	
Parameters	*value*	An mvl value to convert to a four-value logic value.

Converts an mvl value to a subset of four-value logic (see 9.9.1).

Syntax example:

```
var m4 : mvl = mvl_to_mvl4(MVL_U)
```

9.10.8 convert_mvl_list_to_mvl4_list()

Purpose	Convert a list of mvl values to a list of four-value logic subset values.	
Category	Predefined routine	
Syntax	**convert_mvl_list_to_mvl4_list(***mvl-list*: list of mvl**)**	
Parameters	*mvl-list*	A list of mvl values to convert to a list of four-value logic subset values.

This converts a list of mvl values to a list of the four-value logic subset (see 9.9.1). The values in the original list are replaced by the resulting values.

Syntax example:

```
var m4l : list of mvl = {MVL_N; MVL_L; MVL_H; MVL_1};
convert_mvl_list_to_mvl4_list(m4l)
```

9.10.9 mvl_list_to_mvl4_list()

Purpose	Convert a list of mvl values to a list of four-value logic subset values
Category	Predefined routine
Syntax	**mvl_list_to_mvl4_list(***mvl-list*: list of mvl**)**: list of mvl
Parameters	*mvl-list* A list of mvl values to convert to a list of four-value logic subset values.

Converts a list of mvl values to a list of the four-value logic subset (see 9.9.1). A new resulting list is created.

Syntax example:

```
var m4l : list of mvl = mvl_list_to_mvl4_list({MVL_N; MVL_L; MVL_H; MVL_1})
```

9.10.10 string_to_mvl4()

Purpose	Convert a string to a list of four-value logic mvl subset values
Category	Predefined routine
Syntax	**string_to_mvl4(***value-string*: string**)**: list of mvl
Parameters	*value-string* A string representing mvl values, consisting of a width and base followed by a series of characters corresponding to nine-value logic values.

Converts a string into a list of mvl, using the four-value logic subset. Logically, the string is converted to a list of mvl (see 9.9.2), then converted into the four-logic value subset (see 9.9.1).

Syntax example:

```
mlist = string_to_mvl4("8'bxz")
```

9.11 Comparative analysis of ports and tick access

The ***e*** language supports both tick access (see 23.3) and ports in order to access external simulated objects. Ports have the following advantages:

— They support modularity and encapsulation by explicitly declaring interfaces to ***e*** units.

— They are typed.

— They improve the performance of accessing DUT objects with configurable names.

— They can pass not only single values, but also other kinds of information, such as events and queues.

— They can be accompanied in ***e*** with generic or simulator-specific attributes that can be used to specify information needed for enhanced access to DUT objects.

Example 1

This example shows how tick access notation translates to MVL methods, assuming the following numeric port declaration:

```
data : inout simple_port of int is instance;
  keep bind(data, external);
  keep data.hdl_path() == "data";
d: int;
```

```
d = 'data';                    d = data$;

'data' = 32'bz;                data.put_mvl_list(32'bz);

check that 'data@x' == 0;      check that not data.get_mvl_list().has
                                 (it == MVL_X));
                               check that not data.has_x();

d = 'data[31:10]@z'            d = mvl_to_int(data.get_mvl_list(),
                                 {MVL_Z})[31:0]
```

Example 2

This example shows how tick access notation translates to use of an MVL port, assuming the following MVL port declaration:

```
data : inout simple_port of list of mvl is instance;
  keep bind (data, external);
  keep data.hdl_path() == "data";
```

```
check that 'data@x' == 0;      check that not data$.has(it == MVL_X};
                               check that not data.has_x();

'data' = 32'bz;                data$ = 32'bz
```

9.12 *e* port binding

e ports can be bound imperatively by calling **do_bind()** as well as declaratively using the bind syntax defined in **bind()** (9.7.2.1).

9.12.1 do_bind()

Purpose	Connect ports	
Category	Predefined routine	

Syntax	**do_bind(***port-exp1***,** *port-exp2*[**,**…]**);**
	do_bind(*port-exp1***, external);**
	do_bind(*port-exp1***, empty** | **undefined);**	

Parameters	*port-exp1*, *port-exp2*[,…]	One or more expressions of port type. If two expressions are given and the port types are compatible, the two port instances are connected.
	external	Defines a port as connected to an external object.
	empty	Defines a disconnected port. Runtime accessing of a port with an empty binding is allowed.
	undefined	Defines a disconnected port. Runtime accessing of a port with an undefined binding causes an error

Calling the **do_bind()** routine procedurally connects a port to one or more ***e*** ports or to one or more external simulated object. Ports can also be left explicitly disconnected with **empty** or **undefined**.

Syntax example:

```
do_bind(driver.bfm.data_in, bfm.driver.data_out)
```

NOTE 1—The **do_bind()** method can only be called during the **connect_ports()** sub-phase. Calling it at any other time results in an error message.

NOTE 2—It is an error to declare a port disconnected in more than one way.

9.13 Transaction level modeling interface ports in *e*

This clause defines the support the ***e*** language provides for interface ports, used to implement transaction level modeling (TLM) standard interfaces. These ports facilitate the transfer of transactions between verification components, taking advantage of the standardized, high-level TLM communication mechanism.

9.13.1 interface_port

Purpose	Transfer transactions between ***e*** units or a combination of ***e*** units and external modules	
Category	Unit member	
Syntax	*port-instance-name* **:** [**list of**] *direction* **interface_port of** *tlm-intf-type* [**using prefix**=prefix | **using suffix**=suffix] [**is instance**] *port-instance-name* **:** [**list of**] **export interface_port of** *tlm-intf-type* [**is instance**]	
Parameters	*port-instance-name*	A unique ***e*** identifier used to refer to the port or access its value.
	direction	in or out. There is no default.
	tlm-intf-t ype	One of the supported TLM interface types specified in Table 25 or Table 26. The following restrictions apply to the “type” parameter of these interfaces. For internal ***e*** TLM interface ports, the type (or types) specified for the interface shall be any legal ***e*** type. External ***e*** TLM interface ports support transactions of a struct (or class) type only. Thus, for externally bound ***e*** TLM interface ports, the type (or types) specified for the interface shall be legal ***e*** types that inherit from any_struct.
	using prefix=prefix **using suffix**=suffix	Applies for ***e*** TLM input ports only. Specifies a prefix or suffix string to be attached to the predefined TLM methods for the given port. Using a prefix or suffix ensures that there are no method name collisions if a port contains more than instance of an ***e*** TLM interface port tied to the same TLM interface. (This syntax can be used only for the port instance members. It cannot be used in other declarations, such as declarations for parameters or variables.)

An ***e*** TLM interface port type is parameterized with a specific TLM interface type. For example, if an ***e*** TLM interface port is defined with the syntax interface_port of tlm_nonblocking_put, then that port is tied to the tlm_nonblocking_put interface. Then, the set of methods (functions) predefined can be used for that interface to exchange transactions.

Syntax examples:

```
e_packet : in interface_port of tlm_put of packet is instance;


p1 : out interface_port of tlm_nonblocking_transport of
            (packet, msg) is instance;


p2 : export interface_port of tlm_blocking_put is
            instance;  // export
```

9.13.1.1 Special port types

9.13.1.1.1 Export

An export interface port is a port whose enclosing unit does not implement the required interface methods. The interface methods are delegated to the connected unit. An export TLM input port in ***e*** is functionally equivalent to a SystemVerilog or SystemC export.

The following limitations apply to export interface ports:

— The port shall have an outbound connection.

— The port shall be connected (either directly or indirectly) to an input interface port or to an external port providing suitable interface functions.

— The port shall have no inbound connection.

— The port must be connected using the **connect()** (see 9.13.3.2.1) method. The **bind()** constraints and the **do_bind()** routine are not applicable for it.

9.13.1.2 Analysis port

Analysis ports are ports featuring the tlm_analysis interface—a restricted write-only interface intended to share monitoring information for analysis purposes. They may have multiple outbound connections in support of broadcast implementations.

9.13.2 Defining input *e* TLM interface ports

When a unit contains an instance member of an input TLM interface port, the unit must implement all methods required by the TLM interface type of that input port. The list of methods is predefined according to the standard TLM specification.

These methods must be defined before the port is defined. (If the methods and port are defined in the same module, however, the order does not matter.) If any of the required methods is missing, a compile time error shall be issued.

Syntax example:

```
struct packet {
    …
};

unit server {
    // The following four lines define the four methods required
    // by the TLM interface tlm_put.
    put(value : packet)@sys.any is {…};
    try_put(value: packet) : bool is {…};
    can_put() : bool is {…};
    ok_to_put() : tlm_event is {…};
    packet_in : in interface_port of tlm_put of
                   packet is instance;
}
```

In this example, the unit server implements the four methods/tasks that are required by the interface tlm_put of packet.

See the following description of interface method semantics (see 9.13.4.5).

9.13.3 Binding *e* TLM interface ports

9.13.3.1 Binding rules for TLM interface ports

A TLM output port can be bound to a TLM input port if the interface type of the output port is either the same as the interface type of the input port or subset of it (with exactly the same element type in the template parameter). For example, the user can bind an output port of tlm_nonblocking_put to an input port of tlm_put, because the tlm_nonblocking_put interface is a subset of the tlm_put interface. Additionally:

— Empty and undefined bindings are supported for ***e*** TLM interface ports.

— Multiple binding is not supported for ***e*** TLM interface ports, except for analysis ports.

— Unification of ports bound to the same external port is not supported for ***e*** TLM interface ports.

9.13.3.2 Declarative and procedural binding

e TLM interface ports have to be bound before usage, similar to any other port. Binding can be done declaratively with keep **bind()** constraints or procedurally with **do_bind()** or **do_bind_unit()** pseudo-routines. (See 9.7.2.1.)

Syntax examples for declarative binding:

```
keep bind(port1, port2);
keep bind(port3, external)
```

Syntax examples for procedural binding:

```
connect_ports() is also {
  do_bind(port1, port2);
  do_bind(port3, external)
}
```

9.13.3.2.1 connect()—language-neutral binding

External binding of TLM ports in a language-neutral way shall be supported by the simulation environment. The port method **connect()** is provided for this purpose; **connect()** is used to bind two ports that are not both defined in the same language. For example, this method can be used to bind a SystemC port to a SystemVerilog port from ***e***. For uniformity, **connect()** may be used to procedurally bind together ***e*** ports as well.

The **connect()** method shall be called once during the **connect_ports()** phase. The effect of this method is immediate—it shall issue an error in case of any mismatch (wrong external path, mismatching interface types, unsupported multiple binding, and so on).

Syntax of connect():

```
<port1-exp>.connect(<port2-exp>);
<port1-exp>.connect(empty|undefined);
<port1-exp>.connect("external path")
```

Syntax examples for connect():

```
env.agent[1].my_port.connect(env.agent[2].my_export);
env.agent.monitor.port.connect(empty)
```

Description

The following restrictions shall apply to connections created by calling **connect()**.

If port1 is an output port:

— It can be connected to another ***e*** output port, ***e*** export port, or ***e*** input port.

— It can be connected to empty. In this case, this must be the only outbound connection it has. In this case, invoking a method on this port is like calling an empty method.

— It can be connected to undefined. In this case, this must be the only outbound connection it has. In this case, invoking a method on this port will cause to runtime error.

— It can be connected to a specific external port by specifying the external port path. This external port can be of any direction.

If port1 is an export port:

— It can be connected to another ***e*** export port or to other ***e*** input port.

— It can be connected to a specific external port by specifying the external port path. This external port must be an input port or an export port.

Connecting to an external path:

— For SystemVerilog or SystemC, the external path must be quasi-static (full path from the top level scope).

— For ***e***, the external path is an ***e***-path, beginning with “sys.”

9.13.4 Supported TLM interfaces

9.13.4.1 tlm_event predefined struct

The tlm_event predefined struct is used to synchronize a writer and a non-blocking reader on ***e***-to-***e*** TLM ports.

The predefined struct tlm_event is defined as follows:

```
struct tlm_event {
    event trigger;
    notify() is {
       emit trigger
    }
}
```

Some TLM functions return this struct. User code for an input port shall call tlm_event.notify() when it is ready to accept the next transaction. User code performing a non-blocking wait on that input shall be sensitive to emission of the event tlm_event.trigger and shall write the next transaction when that event is emitted.

9.13.4.2 Supported unidirectional TLM interfaces

NOTE—Non-blocking TLM interface calls are zero-delay calls. The blocking interface calls, which correspond to ***e*** TCM methods, consume an additional tick and one cycle of simulation time.

Table 25—Supported TLM interfaces and related methods

TLM interface	Interface methods
Blocking unidirectional interfaces	
tlm_blocking_put of type	put(value:type)@sys.any
tlm_blocking_get of type	get(value:*type)@sys.any
tlm_blocking_peek of type	peek(value:*type)@sys.any
tlm_blocking_get_peek of type	get(value:*type)@sys.any peek(value:*type)@sys.any
Non-blocking unidirectional interfaces	
tlm_nonblocking_put of type	try_put(value:type) : bool can_put() : bool ok_to_put() : tlm_event
tlm_nonblocking_get of type	try_get(value:*type) : bool can_get() : bool ok_to_get() : tlm_event
tlm_nonblocking_peek of type	try_peek(value:*type) : bool can_peek() : bool ok_to_peek() : tlm_event
tlm_nonblocking_get_peek of type	try_get(value:*type) : bool can_get() : bool ok_to_get() : tlm_event try_peek(value:*type) : bool can_peek() : bool ok_to_peek() : tlm_event
Combined unidirectional interfaces (blocking and non-blocking)	
tlm_put of type	put(value:type)@sys.any try_put(value:type) : bool can_put() : bool ok_to_put() : tlm_event
tlm_get of type	get(value:*type)@sys.any try_get(value:*type) : bool can_get() : bool ok_to_get() : tlm_event
tlm_peek of type	peek(value:*type)@sys.any try_peek(value:*type) : bool can_peek() : bool ok_to_peek() : tlm_event

9.13.4.3 Supported bidirectional TLM interfaces

NOTE—Non-blocking TLM interface calls are zero-delay calls. The blocking interface calls, which correspond to ***e*** TCM methods, consume an additional tick and one cycle of simulation time.

Table 26—Supported bidirectional TLM interfaces and related methods

TLM interface	Interface methods
Blocking bidirectional interfaces	
tlm_blocking_master of (req-type, rsp-type)	put(value: req-type)@sys.any get(value: *rsp-type)@sys.any peek(value: *rsp-type)@sys.any
tlm_blocking_slave of (req-type, rsp-type)	put(value: rsp-type)@sys.any get(value: *req-type)@sys.any peek(value: *req-type)@sys.any
tlm_blocking_transport of (req-type, rsp-type)	transport(request: req-type, response: *rsp-type)@sys.any
Non-blocking bidirectional interfaces	
tlm_nonblocking_master of (req-type, rsp-type)	try_put(value: req-type) : bool can_put() : bool ok_to_put() : tlm_event try_get(value: *rsp-type): bool can_get(): bool ok_to_get(): tlm_event try_peek(value: *rsp-type): bool can_peek(): bool ok_to_peek(): tlm_event
tlm_nonblocking_slave of (req-type, rsp-type)	try_put(value: rsp-type) : bool can_put() : bool ok_to_put() : tlm_event try_get(value: *req-type): bool can_get(): bool ok_to_get(): tlm_event try_peek(value: *req-type): bool can_peek(): bool ok_to_peek(): tlm_event
tlm_nonblocking_transport of (req-type, rsp-type)	nb_transport(request: req-type, response: *rsp-type): bool
Combined bidirectional interfaces (blocking and non-blocking)	
tlm_master of (req-type, rsp-type)	put(value: req-type)@sys.any get(value: *rsp-type)@sys.any peek(value: *rsp-type)@sys.any try_put(value: req-type): bool can_put(): bool ok_to_put(): tlm_event try_get(value: *rsp-type): bool can_get(): bool ok_to_get(): tlm_event try_peek(value: *rsp-type): bool can_peek(): bool ok_to_peek(): tlm_event

Table 26—Supported bidirectional TLM interfaces and related methods *(continued)*

TLM interface	Interface methods
tlm_slave of (req-type, rsp-type)	put(value: rsp-type)@sys.any get(value: *req-type)@sys.any peek(value: *req-type)@sys.any try_put(value: rsp-type): bool can_put(): bool ok_to_put(): tlm_event try_get(value: *req-type): bool can_get(): bool ok_to_get(): tlm_event
lm_transport of (req-type, rsp-type)	transport(request: req-type, response: *rsp-type)@sys.any nb_transport(request: req-type, response: *rsp-type): bool

9.13.4.4 Supported analysis TLM interface

Table 27—Supported analysis TLM interface and related methods

TLM interface	Interface methods
tlm_analysis of type	write(value : type)

9.13.4.5 Required semantics of TLM interface methods

TLM interface methods need to be implemented for each interface type. These methods are activated in response to a port interface call. As users of the ***e*** language define new interface types, they will have to provide implementations to the interface methods. The following subclauses define the expected semantics of the various interface methods.

9.13.4.5.1 put(value:type)

The **put()** method passes on a value into the port, making it available for connected ports to read. This call shall block if the port is not ready to handle the transfer of the value.

9.13.4.5.2 try_put(value:type) : bool

The **try_put()** method is non-blocking. If the port is ready to handle a put operation, the value is passed on and the method returns TRUE. Otherwise the method returns FALSE, and the value is not passed on.

9.13.4.5.3 can_put() : bool

The **can_put()** method returns TRUE if the port is ready to handle a put operation [a call to put() will not block]. FALSE is returned if the port is not ready to handle a put operation [a call to put() would block].

9.13.4.5.4 ok_to_put() : tlm_event

The method **ok_to_put()** returns an event that will trigger each time the port is ready to handle a put operation. The returned event may be used to invoke the user code producing the next put operation.

9.13.4.5.5 get(value:*type)

The **get()** method returns the value that is read from the port (the value is passed by reference in the parameter). This call blocks if no value is available to be read.

9.13.4.5.6 try_get(value:*type) : bool

The **try_get()** non-blocking method returns TRUE and the read value if the port can be read. FALSE is returned if the port is not ready to be read [the **get()** operation would block].

9.13.4.5.7 can_get() : bool

The method **can_get()** returns TRUE if a get operation can be performed without blocking. FALSE is returned otherwise.

9.13.4.5.8 ok_to_get() : tlm_event

The method **ok_to_get()** returns an event that will trigger each time the port is ready for a get operation (data is available for reading). The returned event can be used to trigger user code performing a get operation.

9.13.4.5.9 peek(value:*type)

The **peek()** method returns the next value ready to be read from a port. The **peek()** method does not consume the value—a subsequent **get()** call will return the same value. The **peek()** method shall block if no value is ready to be read, and return only when the next value is available.

9.13.4.5.10 try_peek(value:*type) : bool

The **try_peek()** non-blocking method returns TRUE and the read value if the port can be read. FALSE is returned if the port is not ready to be read [the peek() operation would block]. This method does not consume the read value.

9.13.4.5.11 can_peek() : bool

The method **can_peek()** returns TRUE if a peek operation can be performed without blocking. FALSE is returned otherwise.

9.13.4.5.12 ok_to_peek() : tlm_event

The **ok_to_peek()** method returns an event that triggers each time the port is ready for a peek operation (data is available to be read). The returned event can be used to trigger user code that monitors (performs a non-destructive inspection) the ports output.

9.13.4.5.13 transport(request: req-type, response: *rsp-type)

The **transport()** method implements the equivalent of a procedure call, or a bidirectional atomic transfer. The first parameter contains the input to the procedure. The second parameter passes back the output (by reference). The method may consume time, depending on the user-level implementation of this method for a particular interface.

9.13.4.5.14 write(value : type)

The **write()** method passes on a value into an analysis port. The method is non-blocking, because analysis ports should always be ready to be written.

9.14 TLM Sockets in e

This clause defines the support the ***e*** language provides for TLM sockets, used to implement transaction level modeling and communication based on the *IEEE Standard for Standard SystemC® Language Reference Manual (IEEE1666-2011)*. These sockets facilitate the transfer of transactions between verification components, taking advantage of the standardized, high-level TLM 2.0 communication mechanism.

9.14.1 tlm_initiator_socket/tlm_target_socket

Purpose	Provide the interface for accessing external and internal TLM 2.0 sockets.	
Category	Unit member	
Syntax	*tlm-socket-instance* **:** [**list of**] *tlm_socket_type* [**of** *type*] [**using prefix**=*prefix* | **using suffix**=*suffix*] [**is instance**]	
Parameters	*tlm-socket-instance*	A unique ***e*** identifier used to refer to the socket or call any of its access methods/TCMs.
	tlm-socket-type	Defines the socket as an initiator or target socket. Legal values: **tlm_initiator_socket** **tlm_target_socket**
	type	Type of transfer to be used with this socket. Default: **tlm_generic_payload**
	using prefix=prefix **using suffix**=suffix	By default, instantiation of multiple TLM 2.0 sockets in the same unit type results in all sockets using the same predefined methods-because the method names are the same for each socket. To implement a different set of method names for each socket instance, you define a prefix or suffix with using prefix or using suffix. The prefix or suffix you define is attached to all the method names for the current socket instantiation, thus creating a unique set of methods. The prefix or suffix must be a string type, and the text for strings is always enclosed in double quotation marks. (This syntax can be used only for the socket instance members. It cannot be used in other declarations, such as socket reference declarations.)

The tlm_initiator_socket/tlm_target_socket declaration defines and instantiates ***e*** TLM sockets. TLM socket based communication relies on a pair of sockets (one initiator and one target) being connected/bound.

When a unit contains an instance member of a TLM socket, the unit must implement all methods required by the TLM socket type. The list of methods is predefined according to the standard TLM specification.

These methods must be defined before the socket is defined. (If the methods and socket are defined in the same module, however, the order does not matter.) If any of the required methods is missing, a compile time error will be issued.

The phase and time parameters in some of the methods are reference parameters, as designated by the asterisk (*). This enables the method, when called, to update the values of those parameters.

Table 28—TLM socket related methods

	Required interface methods/TCMs
Unit with initiator socket	nb_transport_bw(trans:tlm_generic_payload,phase:*tlm_phase_enum, t:*time):tlm_sync_enum
Unit with target socket	b_transport(trans:tlm_generic_payload, t:*time)@sys.any nb_transport_fw(trans:tlm_generic_payload,phase:*tlm_phase_enum, t:*time):tlm_sync_enum transport_dbg(trans:tlm_generic_payload):uint

e TLM sockets are derived from the any_port base type and thus have the basic facilities of ***e*** ports.

Syntax example:

```
initiator: tlm_initiator_socket of tlm_generic_payload using prefix=
      my is instance;
```

9.14.1.1 nb_transport_bw(trans:tlm_generic_payload, p:*tlm_phase_enum,t:*time):tlm_syn_enum

The **nb_transport_bw()** method is non-blocking and transports a transfer of type tlm_generic_payload (or a type derived from tlm_generic_payload) from the target socket to the initiator socket. **nb_transport_bw()** returns the status of the transaction as a tlm_sync_enum type.

9.14.1.2 nb_transport_fw(trans:tlm_generic_payload, p:*tlm_phase_enum,t:*time):tlm_syn_enum

The **nb_transport_bw()** method is non-blocking and transports a transfer of type tlm_generic_payload (or a type derived from tlm_generic_payload) from the initiator socket to the target socket. **nb_transport_fw()** returns the status of the transaction as a tlm_sync_enum type.

9.14.1.3 b_transport(trans:tlm_generic_payload, ,t:*time)@sys.any

The **b_transport()** TCM is blocking and transports a transfer of type tlm_generic_payload (or a type derived from tlm_generic_payload) from the initiator socket to the target socket.

9.14.1.4 transport_dbg(trans:tlm_generic_payload):uint

The **transport_dbg()** method is non-blocking and gives the initiator socket the ability to read from or write to memory in the target socket. The intent is to provide access to the data for debug purposes.

9.14.1.5 set_bus_width (num:uint)

Predefined method of TLM sockets to set the current bus width. The default is 32 bits.

9.14.1.6 get_bus_width ():uint

Predefined method of TLM sockets to return the current bus width.

9.14.2 Predefined types related to TLM socket based transactions

In accordance with the TLM 2.0 standard there are additional predefined types available in the ***e*** language for handling TLM socket transactions. Details are explained in the following sections.

9.14.2.1 tlm_command

Predefined enumerated type that identifies the basic operation to be performed with the e TLM 2.0 payload. The possible values are `TLM_READ_COMMAND`, `TLM_WRITE_COMMAND` and `TLM_IGNORE_COMMAND`:

— **TLM_READ_COMMAND** – The target socket copies data from the specified address to the transac-tion payload before passing it back to the initiator socket. (In other words, the initiator socket reads data from the target socket.)

— **TLM_WRITE_COMMAND** – The target socket copies data from the current transaction payload to the specified address. (In other words, the initiator socket writes data to the target socket.)

— **TLM_IGNORE_COMMAND** – The target socket does not perform a read or write. It can, however, make use of the value of any attribute in the transaction payload, including extensions. The intent is to allow the payload to act as a vehicle for transporting payload extension values.

9.14.2.2 tlm_endianness

Predefined enumerated type that identifies the endianness of a payload for a TLM transaction. The possible values are `TLM_UNKNOWN`, `TLM_LITTLE_ENDIAN` and `TLM_BIG_ENDIAN`.

9.14.2.3 tlm_extension

Predefined struct type for defining extensions to the tlm_generic_payload struct. The tlm_extension struct is used to define TLM 2.0 data transaction fields that are missing from tlm_generic_payload but that are required to follow a given protocol (or for any other reason). The tlm_extension base struct does not have any public members.

9.14.2.4 tlm_generic_payload

Predefined struct for transferring transaction attributes (like address and data) between TLM sockets. Tlm_generic_payload is inherited of any_sequence_item and can be used in sequence statements. The member field characteristics (e.g. names, sizes) are aligned with the TLM 2.0 standard.

Note on Table 29— For information about how the attributes in this table are used, see the description of the attributes in the IEEE Standard for Standard SystemC® Language Reference Manual (IEEE1666-2011).

Table 29—tlm_generic_payload struct fields

Field Name	
%m_address: uint(bits:64)	Address for the operation.
%m_command: tlm_command	Operation type. See 9.14.2.1.
%m_data: list of byte	Data read or to be written.

Field Name	
%m_length: uint	The number of bytes to be copied to or from the **m_data** array. This field is initialized to zero, which is an invalid value. Thus, this field must be set explicitly when defining the **tlm_generic_payload** struct. To transfer zero data bytes, set **m_command** to TLM_IGNORE_COMMAND. For more information, see the description of TLM_IGNORE_COMMAND in 9.14.2.1.
%m_response_status: tlm_response_status	Status of the operation. See 9.14.2.6.
%m_dmi: bool	DMI stands for direct memory interface. When enabled, it allows the initiator to get direct access to a target memory, bypassing the usual transport interfaces. The default is FALSE.
%m_byte_enable: list of byte	Indicates valid m_data array elements. The default value is zero (null pointer).
%m_byte_enable_length: uint	Indicates the number of elements in the byte enable array. The default value is zero. Note that this attribute is ignored if the value of **m_byte_enable** is zero.
%m_streaming_width: uint	The number of bytes transferred on each beat. The default value is zero. Streaming affects the way a component interprets the data array. A stream consists of a sequence of data transfers occurring on successive notional beats, each beat having the same start address as given by the generic payload address attribute.
%m_extensions: list of tlm_extension	Note This list is empty by default. To add extensions to this list, see the descriptions of **set_extension()** and **get_extension()** in Table 30.

Table 30— tlm_generic_payload Predefined Struct Methods

Method Name	Method Description

set_extension(*ext*: tlm_extension**)**	Adds the specified extension to the generic payload's extension list. Returns the previous value of this extension type, if one existed; otherwise, returns NULL.
	The following example illustrates the use of **set_extension()**:
	`struct extension1 like tlm_extension {`
	`  %m_uint : uint;`
	`  %m_int: int;`
	`  …`
	`};`
	
	`struct extension2 like tlm_extension {`
	`  %m_struct : a_struct;`
	`  …`
	
	`unit initiator_unit {`
	`  i: tlm_initiator_socket is instance;`
	
	`  …`
	`  drive()@sys.any is {`
	`  var gp : tlm_generic_payload;`
	`  gp = new;`
	`  var ext1: extension1 = new;`
	`  …`
	`  gp.set_extension(ext1);`
	`  var ext2: extension2 = new;`
	`  …`
	`  gp.set_extension(ext2);`
	
	`  var status :=`
	`i$.nb_transport_fw(gp,phase,t);`
	`  …`

get_extension(*extension-type-name***)** : tlm_extension	Returns the current value of the specified extension type if such an extension was previously added to the list; otherwise, returns NULL.
get_extensions() : list of tlm_extension	Returns the list of extensions.

9.14.2.5 tlm_phase_enum

Predefined enumerated type to identify the current phase of the communication protocol for non-blocking transport transactions. Possible values are `UNINITIALIZED_PHASE=0`, `BEGIN_REQ=1`,`END_REQ`, `BEGIN_RESP` and `END_RESP`. The values are described below.

UNINITIALIZED_PHASE – No phase has started.

BEGIN_REQ – The request has started.

END_REQ – The request has completed.

BEGIN_RESP – The response has started.

END_RESP – The response has completed.

9.14.2.6 tlm_response_status

Predefined enumerated type to indicate the current response status of a TLM transaction. Possible values and numeric values are `TLM_INCOMPLETE_RESPONSE=0`, `TLM_OKAY_RESPONSE=1`, `TLM_GENERIC_ERROR_RESPONSE=-1`, `TLM_ADDRESS_ERROR_RESPONSE=-2`, `TLM_COMMAND_ERROR_RESPONSE=-3`, `TLM_BURST_ERROR_RESPONSE=-4` and `TLM_BYTE_ENABLE_ERROR_RESPONSE=-5`. These values are described below.

— **TLM_INCOMPLETE_RESPONSE** – The transaction has not yet been delivered to the target or the transaction operation has not yet been executed by the target.

— **TLM_OKAY_RESPONSE** – The transaction operation completed successfully (both read and write operations).

— **TLM_GENERIC_ERROR_RESPONSE** – The operation had an error (can be used by the target to indicate any sort of error).

— **TLM_ADDRESS_ERROR_RESPONSE** – The transaction address is out of range or the operation failed because of the value of the address given in the transaction.

— **TLM_BURST_ERROR_RESPONSE** – An invalid burst was specified. (the target is unable to execute the operation with the given data length).

— **TLM_BYTE_ENABLE_ERROR_RESPONSE** – Either the target does not support byte enables or the value of the byte_enable attribute or the byte_enable_length attribute of the generic payload caused an error.

9.14.2.7 tlm_sync_enum

Predefined enumerated type to identify the synchronization status of non-blocking transactions. Possible values are `TLM_ACCEPTED, TLM_UPDATED` and `TLM_COMPLETED`. These values are described below.

— **TLM_ACCEPTED** – The transaction has been accepted. Neither the transaction object, the phase, nor the delay arguments have been modified.

— **TLM_UPDATED** – The transaction has been modified. The transaction object, the phase, or the delay arguments may have been modified.

— **TLM_COMPLETED** – The transaction execution has completed. The transaction object, the phase, or the delay arguments may have been modified. There will be no further transport calls associated with this transaction.

9.14.3 Binding e TLM sockets

9.14.3.1 Binding rules for TLM sockets

TLM initiator sockets need to be bound to TLM target sockets. Additionally:

— Multiple binding is not supported for e TLM sockets.

— Connection of TLM 2.0 sockets is unidirectional: An initiator socket can connect to its target sock-et, but the target socket cannot connect to the initiator socket.

— The **connect()** method can be called only during the **connect_ports()** or **connect_pointers()** phase.

— Declarative connection using keep **bind()** and procedural connection using **do_bind()** are not sup-ported for sockets.

9.14.3.1.1 connect()-language-neutral binding

Internal and external binding of TLM sockets in a language-neutral way shall be supported by the simulation environment. The port method **connect()** is provided for this purpose; **connect()** is used to bind two sockets.

The **connect()** method shall be called once during the **connect_ports()** phase. The effect of this method is immediate-it shall issue an error in case of any mismatch (wrong external path, mismatching interface types, unsupported multiple binding, and so on).

Syntax of **connect()**:

```
<socket1-exp>.connect(<socket2-exp>);
<socket1-exp>.connect(empty|undefined);
<socket1-exp>.connect("external path")
```

Syntax examples for connect():

```
env.agent[1].i_socket.connect(env.agent[2].t_socket);
env.agent.monitor.i_socket.connect(empty)
```