8. Template types

This clause describes the principles and usage of ***e*** template types. Template types in ***e*** define generic structs and units that are parameterized by type. They can then be instantiated, giving specific types as actual parameters.

NOTE— Because units are a special case of structs, templates can be defined for both structs and units. In general, a template instance is a struct type. Provided it is legal, a template instance becomes a struct as soon as it is referenced. It can be used in any context, and in any way in which a regular struct can be used.

8.1 Defining a template type

Purpose	Define a template type	
Category	Action	
Syntax	[**package**] **template** (**struct** | **unit**) *template-name* **of** **(***param-list***)** [**like** *base-type*] **{**[*member***;**...]**}**	
Parameters	**package**	Denotes **package** access restriction to this template.
	struct or **unit**	Denotes whether the instances of this template are structs or units.
	template-name	The name of the template.
	param-list	A list of (at least one) template type parameters template type parameters, separated by commas. Each parameter is of the form: <[*tag*']**type**> [= *default-type*] *tag* is an optional tag, unique for this template definition. *default-type*, if specified, must be any legal type, possibly deriving from one of the previous parameters. If there is exactly one parameter, the parentheses may be omitted. (see 8.1.1).
	base-type	The base struct from which instances of the template inherit. The base type can itself be parameterized over one or more of the template parameters, in which case each template instance inherits from a different type (see 8.1.2).
	*member***;**...	The body of the template, which contains fields, methods, events, and other struct members. The template parameters can be used within the members as appropriate (see 8.1.3).

NOTE— Template names share the namespace with types, so these names cannot be the same as any other type or template name in the same package. Templates are treated the same as types with respect to the following:

— Name resolution

— Access control

Template definition example

The following example defines a template struct that maps keys to values. The template has two type parameters. The first parameter, <key'type>, is the map key type, and the second parameter, <value'type>, is the value type. When this template is instantiated, the occurrences of the two parameters inside the template body are replaced by the actual types for that instance. The second parameter has a default type: **int**. If the second parameter is not provided at instantiation, the **int** type is used by default.

```
template struct map of (<key'type>, <value'type>=int) {
    keys: list of <key'type>;
    values: list of <value'type>;
    put(k: <key'type>, v: <value'type>) is {
    ...
    };
    get(k: <key'type>): <value'type> is {
    ...
    }
}
```

See 8.2 for an example of how this template would be used.

8.1.1 About template type parameters

A template definition contains a comma-separated list of *template parameters*. A parameter name must have the form <[*tag*']type>, and it can have an optional default type value. A default type can be any legal type. Defaults may be specified only for consecutive parameters at the end of the parameter list. A default may not be specified for a parameter that is followed by a parameter without a default.

The default for a parameter may be derived from one or more previous parameters. For example:

```
template struct map of (<key'type>, <value'type>=<key'type>) {
    ...
}
```

Here the second parameter is, by default, the same type as the first parameter.

When the template is instantiated, a specific type is substituted for each such parameter. Inside the template body, a type parameter can occur at any place where a type is allowed or expected. In the template definition example in 8.1, both <key'type> and <value'type> are used to specify the types of fields, method parameters, and method return values.

8.1.2 Specifying a template base type

A template base type can be a concrete type, so that all of the template instances inherit from the same type, or it can itself be parameterized over template parameters. Examples:

```
struct s { … };
template struct t1 of <type> like s { … };
template struct t2 of <type> like t1 of int { … }
template ordered_set of <type> like map of (<type>,<type>) {…}
```

As for units in general, template units cannot derive from non-unit types, either regular or template.

As for regular structs and units, the default base type for struct templates is **any_struct**, and the default base type for unit templates is **any_unit**.

8.1.3 Template body

The template body consists of struct members. It can contain fields, methods, events, coverage groups, constraints, and any other kind of struct members. A template parameter can be used wherever a type is allowed.

Template body example

```
template struct packet of (<kind'type>, <data'type>) {
    size: uint;
    data1: <data'type>;
    data2: <data'type>;
    kind: <kind'type>;
    keep size < 256;
    sum_data(): <data'type> is {
        return data1 + data2
    }
}
```

8.1.4 Template types and when subtypes

A **when** subtype can be defined within a template in the same way that a **when** subtype can be defined within a regular struct. For example, the following can be added to the packet template in the template definition example in 8.1, assuming that <kind'type> is an enumerated type and the name of one of its items is "red":

Example—when subtype

```
template struct packet of (<kind'type>, <data'type>) {
    ...
    when red {
        red_data: <data'type>
    }
}
```

8.2 Instantiating a template type

Purpose	Instantiate a template	
Category	Type	
Syntax	*template-name* **of (** *actual-param-list* **)**	
Parameters	*template-name*	The name of the template.
	actual-param-list	A comma-separated list of actual type parameters for the template. A legal type name must be specified for each type parameter, but it may be omitted for parameters that have defaults. If only one parameter is specified, the parentheses around the single actual parameter can be omitted. If no parameters are specified, the keyword of can also be omitted.

A template instance creates a new struct, by substituting an actual parameter for the corresponding template parameter within the template body. This substitution also occurs in the definition of the base type.

Parameters at the end of the parameter list may be omitted in the instantiation if defaults exist for them. There is no way in to provide a value to a parameter without providing values to all parameters preceding it. If no value is provided for a parameter, the default is used.

NOTE— All instances of a template are considered to be defined where the template is defined, regardless of where they are instantiated. This applies, in particular, to name resolution and access control issues: each template instance belongs to the package in which the template was declared.

In general, any legal type can be used as an actual type parameter, including another template instance, or another instance of the same template. For example, given the map template in the template definition example in 8.1, the following instances can be created:

```
map of (int, int)
map of (s1, s2)
map of (s, map of (string, int))
map of string
```

Given the packet template in the template definition example in 8.1, the following instances can be created:

```
packet of (color, int)
packet of (color, uint(bits: 64))
```

Not every template instance is legal. A template instance is illegal if, after substituting the actual template parameters in the template body, the template body code becomes illegal. An attempt to refer to an illegal template instance will result in a compilation error. For example, the following two instances are illegal, because the code in the template body implies that <kind'type> must be an enum that has the value “red”:

```
-- Illegal template instances
packet of (int, int)
packet of ([green, blue], int)
```

Any two instances of a template, if their actual parameters refer to exactly the same types, are considered to be the same instance, even if they are syntactically different. For example, in the following code, fields `x` and `y` have the same type:

```
#define N 32;
type color: [red, green];
struct s {
    x: packet of (color, int(bits: 64));
    y: packet of (color, int(bits: N*2))
}
```

Because a template instance becomes a struct, its name can be used anywhere in the code where a struct name is allowed or expected.

8.2.1 Template subtype instances

If a template definition includes **when** subtypes, they are referenced in the same way as regular **when** subtypes. For example, given the packet template in the when subtype example in 8.1.4, the following is a legal type name, because it denotes the red'kind subtype of the “packet of (color, int)” template instance.

```
red packet of (color, int)
```