

6 Template Types

Template types in *e* let you define generic structs and units that are parameterized by type, similar to C++ templates. You can later instantiate them, giving specific types as actual parameters.

The following sections describe the principles and usage of the Specman templates feature:

- “Defining a Template Type” on page 6-1
- “Instantiating a Template Type” on page 6-4
- “Template Types and Reflection” on page 6-6
- “Template Errors ” on page 6-6
- “Limitations” on page 6-8
- “Templates vs. Macros” on page 6-9

Note Because units are a special case of structs, you can define templates 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.

6.1 Defining a Template Type

To define a template, use the **template** statement:

```
[package] template (struct | unit) template-name of (param-list)  
                [like base-type] {[member;...]}
```

package	Denotes package access restriction to this template.
----------------	-------------------------------------------------------------

struct or unit	Denotes whether the instances of this template are structs or units.
------------------------------	----------------------------------------------------------------------

<i>template-name</i>	The name of the template.
<i>param-list</i>	The template parameters (see “About Template Type Parameters” on page 6-3). Currently, only type parameters (<type>) are supported.
<i>base-type</i>	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 “Specifying a Template Base Type” on page 6-3).
<i>member;...</i>	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 “Template Body” on page 6-3).

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:

- Name resolution—See [“Type Name Resolution Rules”](#) on page 10-6 and [“Packages as Namespaces for Types”](#) on page 10-5 in *Usage and Concepts Guide for e Testbenches*
- Access control—[“Controlling Access with Packages”](#) on page 10-3 in *Usage and Concepts Guide for e Testbenches*

Example 1 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.

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

6.1.1 About Template Type Parameters

A template definition contains a comma-separated list of *template parameters*. A parameter name must have the form `<[tag']type>`.

A type parameter can be any legal type in *e*. 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 “[Example 1](#)” on page 6-2, both `<key'type>` and `<value'type>` are used to specify the types of fields, method parameters, and method return values.

6.1.2 Specifying a Template Base Type

A simple way to create a template base type is to specify a concrete type as the base type, so that all of the template instances inherit from the same type. The base type can be a regular struct, or an instance of another template:

```
struct s { ... };
template struct t1 of <type> like s { ... };
template struct t2 of <type> like t1 of int { ... };
```

Another example derives a template from another template:

```
template ordered_set of <type> like set of <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**.

6.1.3 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. You can use a template parameter wherever a type is allowed.

Example 2 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;
    };
};
```

```
};
```

6.1.4 Template Types and when Subtypes

You can define a **when** subtype within a template in the same way that you define a **when** subtype within a regular struct. For example, the following can be added to the packet template in “[Example 2](#)” on page 6-3, assuming that `<kind'type>` has the value “red”:

Example 3 when Subtype

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

6.2 Instantiating a Template Type

Use the following syntax to instantiate a template:

template-name of (*actual-param-list*)

The *actual-param-list* is a comma-separated list of actual parameters for the template. You must give a legal type name for each type parameter. If a template has only one parameter, you can omit the parentheses around the single actual parameter.

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.

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 (see “[Type Name Resolution Rules](#)” on page 10-6 and “[Packages as Namespaces for Types](#)” on page 10-5 in the *Usage and Concepts Guide for e Testbenches*).

In general, you can use any legal type as an actual type parameter, including another template instance, or another instance of the same template. For example, given the map template in “[Example 1](#)” on page 6-2, you can create the following instances:

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

Given the packet template in “[Example 2](#)” on page 6-3, you can create the following instances:

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

Not every template instance is legal. 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 syntactically they are 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.

6.2.1 Template Subtype Instances

If a template definition includes **when** subtypes, you can refer to it in the same way you refer to regular **when** subtypes. For example, given the `packet` template in “[Example 3](#)” on page 6-4, 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)
```

6.2.2 Forward References

The rules for forward referencing of template definitions are the same as for type definitions. A template definition can refer to types, fields, and so on that are not yet defined at that point, but are defined later in the same translation unit¹.

Similarly, a struct or template definition can refer to instances of templates that are defined later in the same translation unit. In particular, a template definition, like a struct definition, can refer to itself in its own body.

1. A single module, or a group of modules that are translated together due to cyclic import.

For example, the following code is legal:

```
template struct t1 of <type> {
    x: t2 of list of <type>;
};
template struct t2 of <type> {
    y: t1 of s;
};
struct s {
    f(): list of t1 of int is { ... };
};
```

6.3 Template Types and Reflection

Template instances, when created, are in fact types, so you can use them in reflection like any other type. With regard to the type name, the canonical name of the instance is used. For example:

```
var rf_s: rf_struct = rf_manager.get_struct_by_name("packet of
(int,int)");
print rf_s.get_name();
rf_s = rf_manager.get_struct_by_name("packet of int(bits:8*2),int");
print rf_s.get_name();
```

prints:

```
packet of (int, int)
packet of (int(bits: 16), int)
```

Note If a template instance has not been used, it does not exist in the type system, so it is not represented in the reflection mechanism.

6.4 Template Errors

Certain kinds of compilation errors are discovered and reported when processing the template definition. Others are discovered and reported upon specific instantiations..

The following sections describe the possible error situations related to templates:

- [“Template Definition Errors” on page 6-7](#)
- [“Template Instantiation Errors” on page 6-7](#)
- [“Run-Time Errors” on page 6-8](#)

6.4.1 Template Definition Errors

Two kinds of compilation errors are discovered and reported when a template definition is processed:

- Syntax errors
- Semantic errors related to type names within the template definition

If these kinds of errors are found within a template definition, they are reported in the same way as when they are in a regular struct definition. In particular, each error message refers to the source line where the error is present.

6.4.2 Template Instantiation Errors

Most kinds of semantic errors are discovered when the template is instantiated. Some instances of the same template can be legal, while others can be illegal and lead to semantic errors, except for those semantic errors related to type names used in the code (“[Template Definition Errors](#)” on page 6-7).

An error message reporting a template instantiation error lists the source lines of the error itself and the source line in which the problematic template instantiation is first created. This report can be recursive—for example, if the template is instantiated within another template, its own instantiation line is also referred to, and so on.¹

For example, trying to load the following code:

```
<'
template struct err_struct of (<first'type>, <second'type>) {
    x: <first'type>(bits: 32); // => <first'type> must be a scalar type
    y: list of <second'type>; // => <second'type> must not be a list type
};
extend sys {
    run() is also {
        var x: err_struct of (string, list of int);
    };
};
'>
```

will cause the following two error messages:

```
*** Error: 'string' is not a scalar type
        at line 3 in err.e
x: <first'type>(bits: 32);
```

1. If any of the code references—the actual erroneous code or one of the template instantiations—originates in a macro, the macro source line is also listed, as for any other macro-related error messages, as described in “[Macro Error Messages](#)” on page 21-35.

```
        from template instantiation 'err_struct of (string, list of int) '  
        at line 8 in err.e  
var x: err_struct of (string, list of int);  
  
*** Error: Cannot define a list of lists  
        at line 4 in err.e  
y: list of <second'type>;  
        from template instantiation 'err_struct of (string, list of int) '  
        at line 8 in err.e  
var x: err_struct of (string, list of int);
```

If several different template instantiations cause errors on the same line of the template definition, all are reported, rather than reporting only one error per line.

6.4.3 Run-Time Errors

A run-time error related to a template definition is reported like any other run-time error. Only the actual error line for interpreted code, or the method containing it for compiled code, is reported, without reference to the template instantiations. This format applies to all kinds of run-time errors, including user errors, DUT errors, assertion failures, and so on.

6.5 Limitations

The following is a list of known limitations related to templates:

- A template instance cannot be created interactively from a command. If a specific instance of a template has not been referenced in the compiled/loaded code, it cannot be used in a command. Trying to do so causes an error message.
- A template instance cannot be created by the reflection mechanism at run-time. If a specific instance of a template has not been referenced in the compiled/loaded code, the reflection mechanism considers it to be a nonexistent type.
- A template method may not be declared as C routine.
- Once a template is defined, it can later—in the same module, or in a later loaded/compiled module—have its direct or indirect base type extended. For some instances, however, the members of the base struct extension might be applied earlier than the members of the original template definition, although they appear later in the code, so the order of the member is incorrect.

Therefore, the behavior for template instances is undefined, and might be different for different instances of the same template. A `WARN_EXTEND_AFTER_TEMPLATE` message is issued when a template instance is created that has the incorrect order for some of its methods, events, or expects. If you get this notification, make sure there is no real problem with this order

Note This behavior might change in the future, so do not rely on it if the order matters.

6.6 Templates vs. Macros

In some sense, templates are similar to define-as macros. Both define parameterized code, which is instantiated by substituting parameters. In principle, a `<statement>` macro can be used for that purpose, if it has `<type>` syntactic arguments and a struct definition in its replacement code. Calling that macro will create a specific struct, which is analogous to creating a template instance. (XREF TO MACRO SECTION)

However, there are several important differences that make using templates preferable:

- A macro must be instantiated explicitly in the code. Multiple explicit instantiations of the same macro causes an error, because the instances are multiple definitions of the same struct. Templates are instantiated automatically, the first time a specific instance is referred to in the code.
- A template definition is fully syntactically analyzed, and any syntax errors within a template definition are reported when the template is defined (“[Template Definition Errors](#)” on page 6-7). A macro definition, in contrast, is not syntactically analyzed, and most syntax errors—except basic errors such as unbalanced parentheses in the code—are discovered only when the macro is called.
- With regard to name resolution and related issues, template code is treated in the context of the template definition, regardless of where a given template instance is referred to for the first time. Macro code, in contrast, is treated in the context of the macro call itself. For instance, if the macro replacement code tries to access the package-accessible field of a struct defined in the same package, the macro call will not compile if it resides in a different package.

