

4. Structs, Subtypes, and Fields

****I've rearrange this clause so all the definitions flow sequentially: struct, subtypes, then fields [see also the notes re: old section 4.10]****

The basic organization of an *e* program is a tree of structs. A *struct* is a compound type that contains data fields, procedural methods, and other members. It is the *e* equivalent of a class in other object-oriented languages. A base struct type can be extended by adding members. Subtypes can be created from a base struct type, which inherit the base type's members, and contain additional members. ****Add a top-level definition for fields here??**

4.1 Structs overview

Structs are used to define data elements and behavior of components of a test environment.

- A struct can hold all types of data and methods.
- All user-defined structs inherit from the predefined base struct type, **any_struct**.
- For reusability of *e* code, use **extend** to add struct members or change the behavior of a previously defined *struct*.

Inheritance is implemented in *e* by using either aspect of a struct definition.

- a) “when” inheritance is specified by defining subtypes with **when** struct members.
- b) “like” inheritance is specified with the **like** clause in new struct definitions.

The best inheritance methodology for most applications is “when” inheritance. See **Annex A** for more information.

4.2 Defining structs: struct

Purpose	Define a data struct
Category	Statement
Syntax	struct <i>struct-type</i> [like <i>base-struct-type</i>] { [<i>struct-member</i> ; ...]}
Parameters	<i>struct-type</i> The name of the new struct type.
	<i>base-struct-type</i> The type of the struct from which the new struct inherits its members.
	<i>struct-member</i> ; ... The contents of the struct. The following are types of struct members: — data fields for storing data — methods for procedures — events for defining temporal triggers — coverage groups for defining coverage points — when , for specifying inheritance subtypes — declarative constraints for describing relations between data fields — on , for specifying actions to perform upon event occurrences — expect , for specifying temporal behavior rules The definition of a struct can be empty, containing no members.

Structs are used to define the data elements and behavior of components and the test environment. Structs contain struct members of the types listed in the *Parameters* description above. Struct members can be conditionally defined (see 4.5).

The optional **like** clause is an inheritance directive. All struct members defined in *base-struct-type* are implicitly defined in the **new** struct subtype, *struct-type*. New struct members can also be added to the inheriting struct subtype and methods of the base struct type can be extended in the inheriting struct subtype.

Additional subtypes can, in turn, be derived from a subtype. In the following example, the subtype `agp_transaction` is derived from the (previously defined) `pci_transaction` subtype. Each subtype can add fields to its base type and place its own constraints on fields of its base type.

Syntax example:

```

type AGPModeType: [AGP_2X, AGP_4X];
struct agp_transaction like pci_transaction {
    block_size: uint;
    mode: AGPModeType;
    when AGP_2X agp_transaction {
        keep block_size == 32;
    };
    when AGP_4X agp_transaction {
        keep block_size == 64;
    };
};

```

4.3 Extending structs: extend type

Purpose	Extend an existing data struct	1
Category	Statement	5
Syntax	extend [<i>struct-subtype</i>] <i>base-struct-type</i> { [<i>struct-member</i> ; ...]}	10
Parameters	<i>struct-subtype</i> Adds struct members to the specified subtype of the base struct type only. The added struct members are known only in that subtype, not in other subtypes.	
	<i>base-struct-type</i> The base struct type to extend.	15
	<i>struct-member</i> ; ... <ul style="list-style-type: none"> — data fields for storing data — methods for procedures — events for defining temporal triggers — coverage groups for defining coverage points — when, for specifying inheritance subtypes — declarative constraints for describing relations between data fields — on, for specifying actions to perform upon event occurrences — expect, for specifying temporal behavior rules The definition of a struct can be empty, containing no members.	20 25

This adds struct members to a previously defined struct or struct subtype. Members added to the base struct type in extensions apply to all other extensions of the same struct, e.g., if a method in a base struct is extended using **is only**, it overrides that method in every one of the **like** children.

If **like** inheritance has been used on a struct type, there are limitations on further extending the original base struct type definition; see the following subsections.

Syntax example:

```

type packet_kind: [atm, eth];
struct packet {
    len: int;
    kind: packet_kind;
};
extend packet {
    keep len < 256;
};

```

4.3.1 Restrictions due to inherent differences

Some of the restrictions on **like** inheritance derive from the inherent differences between **when** and **like** inheritance.

- Determinant fields shall not be explicitly referenced.
- Creating multiple, orthogonal subtypes can be difficult with **like** inheritance.
- Generation of a parent does not create **like** children.
- **when** subtypes shall not be added to a *struct* with **like** children. Similarly, a **like** child shall not be created from a *struct* that has **when** subtypes.

4.3.2 Generation restrictions on like inheritance

This section describes restrictions on generation when **like** inheritance is used.

- Temporary fields in the parent can cause problems. Constraints that have expressions on one side of an equality or inequality create temporary fields. For example:

```
keep a > b * c;
```

gets translated internally into:

```
keep tmp == b * c; keep a > tmp;
```

If such constraints are specified in a parent, this can cause a crash during runtime. (There is no problem with constraints in a leaf child.)

- Unidirectional constraints in the parent do not induce generation order. Unidirectional constraints in the parent struct do not induce the expected generation order in the child. For example, suppose that the following constraint appears in packet:

```
keep size == f(b);
```

During generation of a like-inherited packet struct, the constraint above does not cause `b` to be generated before `size`. This often leads to a contradiction.

4.4 Extending subtypes

A *struct subtype* is an instance of the struct in which one of its fields has a particular value. For example, the `packet` struct defined in the following example has `atm packet` and `eth packet` subtypes, depending on whether the `kind` field is `atm` or `eth`.

Example

```
type packet_kind: [atm, eth];
struct packet {
    len: int;
    kind: packet_kind;
};
extend packet {
    keep len < 256;
};
```

Similar to structs, a struct subtype can be **extended**; the extension shall only apply to that subtype.

4.5 Creating subtypes with when

The **when** struct member creates a conditional subtype of the current struct type when a particular field of the struct has a given value. This is called “when” inheritance and is one of two techniques that *e* provides for implementing inheritance. The other is called “like” inheritance. When inheritance is described in this section. Like inheritance is described in 4.2.


```

1      } ;
      };

```

4.6 Extending when subtypes

There are two general rules governing the extensions of when subtypes:

- a) If a struct member is declared in the base struct, it shall not be re-declared in any **when** subtype, but it can be extended.
- b) With the exception of coverage groups and the events associated with them, any struct member defined in a **when** subtype does not apply or is unknown in other subtypes, including:
 - 1) fields
 - 2) constraints
 - 3) events
 - 4) methods
 - 5) **on**
 - 6) **expect**
 - 7) **assume**

4.6.1 Coverage and when subtypes

All coverage events shall be defined in the base struct. Attempts to do so within a subtype, however, shall result in a load time error. Coverage groups shall be defined in the base struct or in the subtype.

4.6.2 Extending methods in when subtypes

A method defined or extended within a **when** construct is executed in the context of the subtype and can freely access the unique struct members of the subtype with no need for any casting.

When a method is declared in a base type, each extension of the method in a subtype shall have the same parameters and return type as the original declaration. Attempts to do otherwise shall result in a load time error. However, if a method is not declared in the base type, each definition of the method in a subtype can have different parameters and return type.

If more than one method of the same name is known in a **when** subtype, any reference to that method is ambiguous and shall result in a load-time error. To remove the ambiguity from such a reference, use the **as_a()** type casting operator or the **when** subtype qualifier syntax.

****Add x-ref to the as_a() operator****

Example 1

```

45      p.as_a(legal packet).show();
      break on call legal packet.show()

```

Method calls are checked when the e code is parsed. If there is no ambiguity, the method to be called is selected and all similar references are resolved in the same manner.

In the example above, the extension to ethernet packet could be placed in a separate file like this:

Example 2

```

55      extend packet {

```

```

    when ethernet packet {
        show() is {out("it is a ethernet packet")};
    };
};

```

If this file is loaded after the rest of the *e* code has been loaded, no error is issued because the method call to `p.show()` was resolved when the first file was loaded. Any call to `p.show()` always prints:

```
it is a legal packet
```

4.7 Defining fields: field

Purpose	Define a struct field	
Category	Struct member	
Syntax	[!][%] <i>field-name</i> [: <i>type</i>] [<i>min-val</i> .. <i>max-val</i>] [(bits bytes): <i>num</i>]	
Parameters	!	Denotes an ungenerated field. The “!” and “%” options can be used together, in either order.
	%	Denotes a physical field. The “!” and “%” options can be used together, in either order.
	<i>field-name</i>	The name of the field being defined.
	<i>type</i>	The type for the field. This can be any scalar type, string, struct, or list. If the field name is the same as an existing type, the “: <i>type</i> ” part of the field definition can be omitted. Otherwise, the type specification is required.
	<i>min-val..max-val</i>	An optional range of values for the field, in the **form?? . If no range is specified, the range is the default range for the field’s type.
	(bits bytes : <i>num</i>)	The width of the field in bits or bytes. This syntax allows you to specify a width for the field other than the default width. This syntax can be used for any scalar field, even if the field has a type with a known width.

This defines a field to hold data of a specific type. It can be a physical field (%) or a virtual field (the default), and automatically generated (!) or not (the default). For scalar data types, the size of the field can also be specified in bits or bytes.

4.7.1 Physical fields

A field defined as a physical field (with the % option) is packed when the struct is packed. Fields that represent data to be sent to the HDL device in the simulator or that are to be used for memories need to be physical fields. Nonphysical fields are called *virtual fields* and are not packed automatically when the struct is packed, although they can be packed individually.

If no range is specified, the width of the field is determined by the field’s type. For a physical field, use the **(bits : num** or **bytes : num)** syntax to specify the width when the field’s type does not have a known width.

4.7.2 Ungenerated fields

A field defined as ungenerated (with the `!` option) is not generated automatically. This is useful for fields that are to be explicitly assigned during a test or whose values involve computations that cannot be expressed in constraints.

Ungenerated fields have default initial values (0 for scalars, `NULL` for structs, and an empty list for lists). An ungenerated field whose value is a range (such as `[0..100]`) gets the first value in the range. If the field is a *struct*, ****the struct??** is not allocated and none of the fields in it are generated.

4.7.3 Assigning values to fields

Unless a field is defined as ungenerated, a value is generated for it when the *struct* is generated, subject to any constraints that exist for the field. However, even for generated fields, values can be assigned in user-defined methods or predefined methods, such as `init()`, `pre_generate()`, or `post_generate()`. The ability to assign a value to a field is not affected by either the `!` option or any generation constraints.

Syntax example:

```
type NetworkType: [IP=0x0800, ARP=0x8060] (bits: 16);
struct header {
    address: uint (bits: 48);
    hdr_type: NetworkType;
    !counter: int;
};
```

4.8 Defining list fields

This section defines lists.

4.8.1 list of

Purpose	Define a list field	
Category	Struct member	
Syntax	[!][%] <i>list-name</i> [<i>length-exp</i>]: list of <i>type</i>	
Parameters	<code>!</code>	Do not generate this list. The “!” and “%” options can be used together, in either order.
	<code>%</code>	Denotes a physical list. The “!” and “%” options can be used together, in either order.
	<i>list-name</i>	The name of the list being defined.
	<i>length-exp</i>	An expression that gives the initial size for the list. The expression shall evaluate to a non-negative integer.
	<i>type</i>	The type of items in the list. This can be any scalar type, string, or struct. It shall not be a list.

This defines a list of items of the specified type.

An initial size can be specified for the list; the list initially contains that number of items. The size shall conform to the initialization rules, the generation rules, and the packing rules. Even if an initial size is specified, the list size can change during a test if the list is operated on by a list method that changes the number of items.

All list items are initialized to their default values when the list is created. For a generated list, the initial default values are replaced by generated values. For information about initializing list items to particular values, see 3.1.4.3.6 and 7.1.7.

Syntax example:

```
packets: list of packet;
```

4.8.2 list(key) of

Purpose	Define a keyed list field	
Category	Struct member	
Syntax	![%]list-name: list(key: key-field) of type	
Parameters	!	Do not generate this list. For a keyed list, the “!” is required, not optional.
	%	Denotes a physical list. The “%” option can precede or follow the “!”.
	list-name	The name of the list being defined.
	key-field	The key of the list. For a list of structs, it is the name of a field of the struct. For a list of scalar or string items, it is the item itself, represented by the it variable. This is the field or value which the keyed list pseudo-methods check when they operate on the list.
	type	The type of items in the list. This can be any scalar type, string, or struct. It shall not be a list.

Keyed lists are used to enable faster searching of lists by designating a particular field or value to use during the search. A keyed list can be used, for example, in the following ways:

- as a hash table, in which searching only for a key avoids the overhead of reading the entire contents of each item in the list.
- for a list that has the capacity to hold many items, but only contains a small percentage of its capacity, randomly spread across the range of possible items. An example of this is a sparse memory implementation.

Besides the **key** parameter, the keyed list syntax differs from the regular list syntax in the following ways.

- a) The list shall be declared with the **!** do-not-generate operator. This means a keyed list needs to be built item-by-item, since it cannot be generated.
- b) The *[exp]* list size initialization syntax is not allowed for keyed lists, i.e., *list[exp]: list(key: key) of type* is not legal. Similarly, **keep** shall not be used to constrain the size of a keyed list.
- c) A keyed list is a distinct type, different from a regular list. This means a keyed list can be assigned to a regular list or vice-versa, e.g., if *list_a* is a keyed list and *list_b* is a regular list, *list_a = list_b* results in a syntax error.

If the same key value exists in more than one item in a keyed list, the keyed list pseudo-methods use the latest item in the list (the one with the highest list index number). Other items with the same key value are ignored. The keyed list pseudo-methods (see 19.8) only work on lists that were defined and created as keyed lists. Conversely, restrictions apply when using regular list pseudo-methods or other operations on keyed lists (see 19.9).

Syntax example:

```
!locations: list(key: address) of location;
```

4.9 Defining attribute fields

Purpose	Define the behavior of a field when copied or compared
Category	Unit member
Syntax	attribute <i>field-name</i> <i>attribute-name</i> = <i>exp</i>
Parameters	<i>field-name</i> The name of a field in the current struct.
	<i>attribute-name</i> <i>attribute-name</i> is one of the following: <ul style="list-style-type: none"> a) <code>deep_copy</code> — controls how the field is copied by the deep_copy() routine. b) <code>deep_compare</code> — controls how the field is compared by the deep_compare() routine. c) <code>deep_compare_physical</code> — controls how the field is compared by the deep_compare_physical() routine. d) <code>deep_all</code> — controls how the field is copied by the deep_copy() routine or compared by the deep_compare() or deep_compare_physical() routines.
	<i>exp</i> <i>exp</i> is one of the following: <ul style="list-style-type: none"> a) <code>normal</code> — performs a deep (recursive) copy or comparison. b) <code>reference</code> — performs a shallow (non-recursive) copy or comparison. c) <code>ignore</code> — do not copy or compare.

Defining attributes controls how a field behaves when it is copied or compared. These attributes are used by **deep_copy()**, **deep_compare()**, and **deep_compare_physical()**. For a full description of the behavior specified by each expression, see 24.1.1, 24.1.2, or 24.1.3 respectively.

To determine which attributes of a field are valid, all extensions to a unit or a *struct* are scanned in the order they were loaded. If several values are specified for the same attribute of the same field, the last attribute specification loaded is the one that is used.

The **attribute** construct can appear anywhere, including inside a **when** construct or an **extend** construct.

Syntax example:

```
attribute channel deep_copy = reference;
```

4.10 Comparison of when and like inheritance

****I've made this section into an Informative Annex [Annex A]
(except for parts of 4.10.6, which I've appended to section 4.3)****

There are two ways to implement object-oriented inheritance in e:

- Like inheritance is the classical, single inheritance familiar to users of all object-oriented languages.
- When inheritance is a concept introduced by e. It is less familiar initially, but lends itself more easily to the kind of modeling done in e.

This section discusses the pros and cons of both these types of inheritance and recommends when to use each of them.

1
5
10
15
20
25
30
35
40
45
50
55

1

5

10

15

20

25

30

35

40

45

50

55