

## Annex A 1

(informative)

### Comparison of when and like inheritance 5

There are two ways to implement object-oriented inheritance in *e*: 10

- 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 annex discusses the pros and cons of both these types of inheritance and recommends when to use each of them. 15

#### A.1 Summary of when versus like 20

In general, “when” inheritance should be used for modeling all DUT-related data structures. It is superior from a knowledge representation point of view and from an extensibility point of view. When inheritance can:

- explicitly reference a field that determines the **when** subtype 25
- create multiple, orthogonal subtypes
- use random generation to generate lists of objects with varying subtypes
- easily extend the *struct* later.

Although like inheritance has more restrictions than when inheritance, it is recommended in some special cases because: 30

- a) Like inheritance is somewhat more efficient than when inheritance.
- b) Generation of objects that use like inheritance can also be more efficient. 35

##### A.1.1 A simple example of when inheritance

A **when** subtype of a generic struct can be created using any field in the *struct* that is a Boolean or enumerated type. This field, which determines the **when** subtype of a particular struct instance, is called the *when determinant*. In the following example, the when determinant is `legal`. 40

*Example 1*

```

struct packet { 45
    legal: bool;

    when legal packet {
        pkt_msg() is {
            out("good packet");
        };
    };
};

```

NOTE—The following syntax is used in this document because it looks closer to the “like” version: 55

```
1      extend legal packet {...}
```

This syntax is exactly equivalent to the **when** construct:

```
5      extend packet {when legal packet {...}}
```

The following example shows a generic packet struct with three fields: `protocol`, `size` and `data`, and an abstract method `show()`. In this example, the “protocol” field is the determinant of the when version of the packet, i.e., this field determines whether the packet instance has a subtype of “IEEE”, “Ethernet”, or “foreign”. In this example, the Ethernet packet subtype is extended by adding a field and extending the `show()` method.

### 15 Example 2

```

type packet_protocol: [Ethernet, IEEE, foreign];
struct packet {
    protocol: packet_protocol;
    size: int [0..1k];
    data[size]: list of byte;
    show() is undefined; // To be defined by children
};
extend Ethernet packet {
    e_field: int;
    show() is {out("I am an Ethernet packet")};
};

```

Of course, it is possible for a *struct* to have more than one when determinant. In the following example, the Ethernet packet subtype is extended with a field of a new enumerated type, `Ethernet_op`.

### 30 Example 3

```

type Ethernet_op: [e1, e2, e3];
extend Ethernet packet { op: Ethernet_op; };
extend e1 Ethernet packet {
    e1_foo: int;
    show() is {out("I am an e1 Ethernet packet")};
};

```

Because it is possible for a *struct* to have more than one when determinant, the inheritance tree for a struct using when inheritance consists of any number of orthogonal trees, each rooted at a separate enumerated or Boolean field in the struct. Figure A1 shows a when inheritance tree consisting of three orthogonal trees rooted in the `legal`, `protocol`, and `op` fields.

NOTE—The **when** subtypes that have not been explicitly defined, such as IEEE packet, exist implicitly.

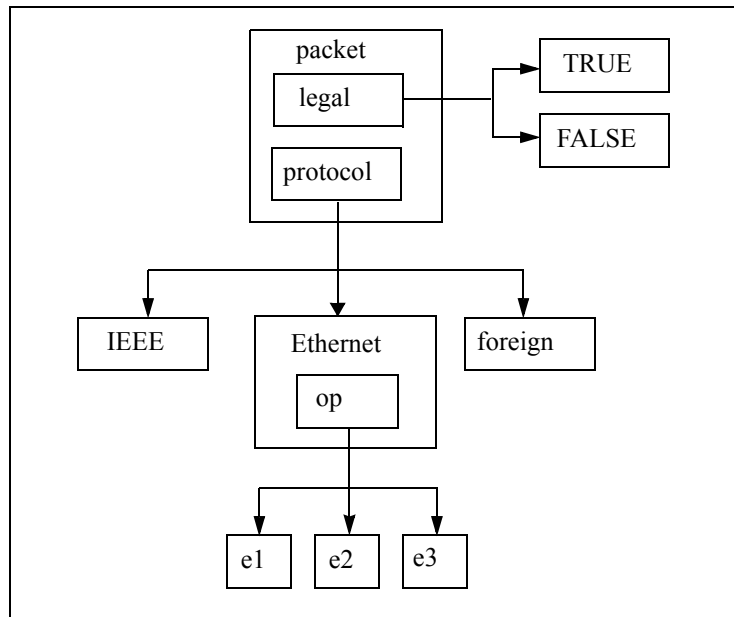


Figure A1—When inheritance tree for packet struct subtypes

### A.1.2 A simple example of like inheritance

A like child of a generic struct can be created by using the **like** construct. In this example, a child `Ethernet_packet` is created from the generic struct `packet` and is extended by adding a field and extending the `show()` method.

#### Example

```

struct packet {
    size: int [0..1k];
    data[size]: list of byte;
    show() is undefined; // To be defined by children
};
struct Ethernet_packet like packet {
    e_field: int;
    show() is {out("I am an Ethernet packet")};
};
  
```

In the same way, an `IEEE_packet` can be created from `packet` using **like**:

```

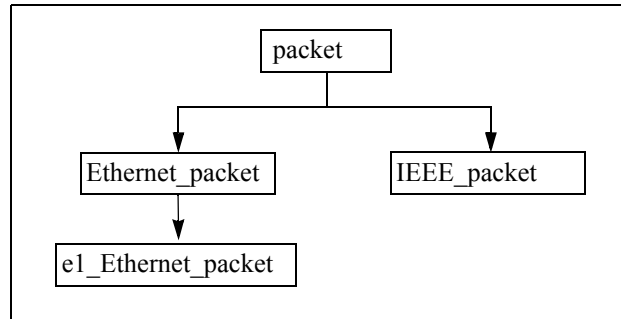
struct IEEE_packet like packet {
    i_field: int;
    show() is {out("I am an IEEE packet")};
};
  
```

Or an `e1_Ethernet_packet` can be created from `Ethernet_packet` using like inheritance.

```

struct e1_Ethernet_packet like Ethernet_packet {
    e1_foo: int;
    show() is {out("I am an e1 Ethernet packet")};
};
  
```

In contrast to the when inheritance tree, the like inheritance tree for the packet type is a single tree where each subtype needs to be defined explicitly, as shown in Figure A2. This difference between the like and when inheritance trees is the essential difference between like and when inheritance.



**Figure A2—Like inheritance tree for packet struct subtypes**

## A.2 Advantages of using when inheritance for modeling

While the like version and the when version look similar, and the “like” version may seem more natural to those familiar with other object-oriented languages, the “when” version is much better for the kind of modeling typically done in e. There are several reasons for this:

- Determinant fields can be explicitly referenced
- Multiple orthogonal subtypes can be used
- Lists of objects with varying subtypes can be used
- The struct can be extended later
- A new type can be created by simple extension

### A.2.1 Determinant fields can be explicitly referenced

In the when version, the determinant of the when is an explicit field. In the like version, there is no explicit field that determines whether a packet instance is an Ethernet packet, an IEEE packet, or a foreign packet. The explicit determinant fields provide several advantages:

- Explicit determinant fields are more intuitive.  
Fields are more tangible than types and correspond better to the way hardware engineers perceive architectures. Having a field whose value determines what fields exist under it is familiar to engineers. (It is similar to C unions, for example.)
- Attributes of determinants that are physical fields can be used.  
If the determinant is a physical field, it might be desirable to specify its size in bits, the mapping of enumerated items to values, where it is in the order of fields, and so on. These things are done very naturally with when inheritance, because the determinant is just another field. For example:
 

```
%protocol: packet_protocol (bits: 2);
```
- With like inheritance, a field can be defined as the when determinant is, but it also needs to be tied into the type with code similar to the following:

```

var pkt: packet;
case protocol {
  Ethernet {var epkt: Ethernet packet; gen epkt; pkt = epkt;};
  IEEE {var ipkt: IEEE packet; gen ipkt; pkt = ipkt;};
};

```

Plus, there is an added inconvenience of having to generate or calculate protocol separately from the rest of the packet.

- The when determinant can be constrained.

Using when inheritance, it is very natural to write constraints like these in a test:

```

keep protocol in [Ethernet, IEEE];
keep protocol != IEEE;
keep soft protocol == select { 20: IEEE; 80: foreign; };
keep packets.is_all_iterations(.protocol, ...);

```

Constraining the value of fields in various ways is a main feature of generation. Doing the same with like inheritance is more complicated. For example, the first constraint above might be stated something like this:

```

keep me is an Ethernet_packet or me is an IEEE_packet;
// This pseudocode is not a legal constraint specification

```

However, constraints like this can become quite complex in like inheritance. Furthermore, there is no way to write the last two constraints.

## A.2.2 Multiple orthogonal subtypes can be used

Suppose each packet (of any protocol) can be either a normal (data) packet, an ack packet, or a nack packet, except that foreign packets are always normal.

*Example*

```

type packet_kind: [normal, ack, nack];
extend packet {
  kind: packet_kind;
  keep protocol == foreign => kind == normal;
};
extend normal packet { n1: int; };

```

How can this be done in like inheritance (disregard for now the issue of extending the packet struct later)? Assuming the requirement stated above is known in advance and it should be modeled using like inheritance in the best possible way:

```

struct normal_Ethernet_packet like Ethernet_packet {
  n1: int;
};
struct ack_Ethernet_packet like Ethernet_packet { ... };
struct nack_Ethernet_packet like Ethernet_packet { ... };
struct normal_IEEE_packet like IEEE_packet { ... };

```

This requires eight declarations.

Then, the `Ethernet_op` possibilities need to be taken into account:

```
1      struct ack_e1_Ethernet_packet like e1_Ethernet_packet { ... }
```

This works, but requires  $((N_1 * N_2 * \dots * N_d) - \text{IMP})$  declarations, where  $d$  is the number of orthogonal dimensions,  $N_i$  is the number of possibilities in dimension  $i$ , and  $\text{IMP}$  is the number of impossible cases.

Another issue is how to represent the impossible cases.

Multiple inheritance would solve some of these problems, but would introduce new complications.

With when inheritance, all the possible combinations exist implicitly, but they do not have to be enumerated. It is only when something needs to be specified about a particular one that it is enumerated, as in the following examples:

```
15      extend normal IEEE packet { ni_field: int; }; // Adds a field
      extend ack e1 Ethernet packet { keep size == 0; }; // Adds a constraint
```

All in all, the when version is more natural from a knowledge representation point of view, because:

- It is immediately clear from the description what goes with what.
- Types only need to be specified if there is something to say about them.

### A.2.3 Lists of objects with varying subtypes can be used

The job of the generator is to create (in this example) packet instances. By default, all possible packets are generated. In both versions, a list of packets is created, e.g.,

```
      extend sys { packets: list of packet; };
```

However, the generator should only generate fully instantiated packets. In the when version, that happens automatically — there is no other way.

With like inheritance, when a parent struct is generated, only that parent struct is created; none of the like children are created. For example, the following gen action always creates a generic packet, never an Ethernet packet or an IEEE packet:

```
      pkt: packet;
      gen pkt;
```

Thus, in practice, only fields whose type is a leaf in the like inheritance tree should be generated, e.g.,

```
      p: e1_Ethernet_packet;
      gen p;
```

### A.2.4 The struct can be extended later

There are some restrictions on extending structs that have like children, however, see 4.3 for more details.

### A.2.5 A new type can be created by simple extension

The following example extends the packet\_protocol type and adds new members to the packet subtype.

```
55      extend packet_protocol: [brand_new];
      extend brand_new packet {
```

```

    ...new struct members...
};

```

1

The old environment is automatically able to generate the `brand_new` packets. With like inheritance, all instances of the procedural generation code need to be identified first and then a new case needs to be added to the case statement.

5

### A.3 Advantages of using like inheritance

10

Like inheritance is a shorthand notation for a subset of when inheritance. It is restricted, but more efficient. Like inheritance often has better performance than when inheritance for the following reasons:

- Method calling is faster for like inheritance.
- When generation is slower than like generation. This can be important if a large part of the total runtime is attributable to generation.
- When inheritance uses more memory because all of the fields of all of the when subtypes consume space all the time.

15

20

NOTE—If this becomes a problem in a particular design, there is a workaround. Rather than having many separate fields under the **when**, put all the fields into a separate *struct* and put a single field for that struct under the **when**. For example, the following coding style could use a lot of memory if there are many fields declared under the Ethernet packet subtype.

```

type packet_protocol: [Ethernet, IEEE, foreign];
struct packet {
    protocol: packet_protocol;
    when Ethernet packet {
        e_field0: int;
        e_field1: int;
        e_field2: int;
        e_field3: int;
        // ...
    };
};

```

25

30

35

A more efficient coding style is shown below, where a single field is declared under the Ethernet packet subtype.

```

type packet_protocol: [Ethernet, IEEE, foreign];
struct Ethernet_packet {
    e_field0: int;
    e_field1: int;
    e_field2: int;
    e_field3: int;
    // ...
};
struct packet {
    protocol: packet_protocol;
    when Ethernet packet {
        e_packet: Ethernet_packet;
    };
};

```

40

45

50

55

## A.4 When to use like inheritance

Like inheritance should be used for modeling only when the performance win is big enough to offset the restrictions, for example:

- objects that use a lot of memory, such as a register file, where the number of distinct registers is very large, and for each such register a field of the register type needs to be generated, e.g., “pc: pc\_reg”, “psr: psr\_reg”, and so on.
- objects that do not require randomization, such as a scoreboard or a memory.

Like inheritance should also be used for non-modeling, programming-like activities, such as implementing a generic package for a queue.

**\*\*I have removed the rest of this (examples of like inheritance restrictions and when inheritance)\*\***