

Make Generation As Sequential As Possible

Generation with static constraints seem to be an elegant and quite comfortable way of writing, which makes one use it everywhere, without thinking whether it is really needed. Then you are bound to discover two of Specman's most problematic points, Generation often does not work as expected with no apparent reason, and the so called generation debugger doesn't help a lot to find out why.

Another method that effectively overcomes the above mentioned problem is Generation using sequential code. Let us see both the approaches used in generation . Here the same code first is written using static constraints, and then rewritten with sequential generation.

1. Generation using static constraints:

```
<'
type ei_gbe_packet_type : [E1,E2];
//This line defines an enumerated type ei_gbe_packet_type which can take the values E1,E2

struct ei_gbe_packet {
    packet_type : ei_gbe_packet_type;
    %header : byte; // first header byte of the packet
    keep packet_type == E1 => header != 0;
    keep packet_type == E2 => header == 100;

    %data_length : byte; //data length of packet
    keep (header > 20) and (packet_type == E1) => data_length ==46;
    keep (header <120) and (packet_type != E2) => data_length ==15;
};
'>
```

2. With sequential generation:

```
<'
type ei_gbe_packet_type : [E1,E2];

struct ei_gbe_packet {
    packet_type : ei_gbe_packet_type;

    !%header : byte;
    // first header byte of the packet, the exclamation means that it should not be generated automatically
    as the struct is created

    !%data_length : byte;

    post_generate() is also{
        case (packet_type) {
            E1 : {
                gen header keeping {it != 0};
            };
            E2: {
                gen header keeping {it == 100};
            };
        };
    };
};
```

```
};
};

if (header > 20 and packet_type == E1) {
    gen data_length keeping {it == 46};
}else{
    if (header < 120 and packet_type != E1) {
        gen data_length keeping {it == 15};
    }else{
        gen data_length;
    };
};
};
};
';
```

It is important to note that only the field `packet_type` is assigned a random value as the packet is created. The other fields are prefixed with an exclamation mark, which means that they should not be generated, in other words, should not be assigned a random value. So `packet_type` gets to be E1,E2 while `header` and `data_length` retain a garbage value, which is supposed to be zero. After the packet is created and all fields which are not prefixed with an exclamation mark are assigned a random value (in our case only `packet_type`). The method `post_generate()` is a predefined method of every struct which enables user to do some specific operations after the struct is created.

During this phase, Specman takes all the constraints on a specific field and tries to solve them, in other words, to find a group of values that does not contradict any of the constraints. This type of constraints does not exist in the sequential version, since the 'if' and 'case' statements already take care of all these dependencies.