Progressive Coverage is a method of coverage collection that’s highly applicable in communication devices, but may as well be applied elsewhere. Now before I start to babble about my philosophy on coverage collection why don't I just give you an example?

Let’s recall our beloved struct from a recent post - packet - and make it a bit more interesting:

{code}

```
struct packet {
 %header: byte;
 %payload: list of byte:
 packet_valid: bool;
 header_crc_ok: bool;
 payload_crc_ok: bool;
 };
```

{/code}

Our sample packet can now have various types of errors. First off, it may be invalid (as indicated by the packet_valid field). If the frame is valid, it might have a corrupt header (header_crc_ok == FALSE), in which case it would be impossible to parse the payload because whoever receives this packet doesn’t know what kind of packet it is. But that’s not all folks, the plot thickens. Even if the header is ok, the payload itself might still contain errors, in which case the payload_crc_ok would be FALSE. Got it?

Ok, so now we want to write a reference model for the Receiver part which - how surprisingly - receives packets of that particular type, tries to parse them and then do something with them - not so important. What I’d like to focus on today is the coverage collection **within** the reference

model.

So let’s get started!

Roughly, the reference model will look something like this:

{code}

```
unit rx_ref_model {
 !curr_pkt: packet; // currently processed packet
 event cover_curr_pkt; // coverage event
 // method port that receives a packet from somewhere
 receive_packet: in method_port of packet_type is instance;
 // implementation of the receiving method
 receive_packet(pkt: packet) is {
 curr_pkt = pkt; // copy pkt to be used locally
 if pkt.packet_valid then { process_header(); };
 };
 process_header() is {
 // do something and then
 if pkt.header_crc_ok then { process_payload() };
 };
 process_payload() is {
 // do something
 if pkt.payload_crc_ok then { // do something; };
 };
 };
```

{/code}

So this is the skeleton of the reference model, and the important thing to notice is that the nature of the processing is progressive - you first process the packet as a whole, then the header, and last - the payload. And now ladies and gentlemen, we must add some coverage. We’ll make it really simple and assume we want coverage on the header value, and the payload

size.

{code}

extend rx_ref_model {

cover cover_curr_pkt is {
 item header;
 item payload_size: uint = payload.size();
 receive_packet() is also {
 emit cover_curr_pkt;
 };
};{/code}

So now we cover all we want upon receipt of each packet in the reference model. But is that all? Of course not. What's the point of covering the payload size if the header is invalid? So let's use the WHEN command to make sure we only cover items when relevant.

{code}
item header using when = packet_valid and header_crc_ok;
 item payload_size using when=packet_valid and header_crc_ok and payload_crc_ok;
{/code}

Now the coverage will only be collected when relevant. Great! we're finished with the introduction. Phew

In modern network devices we might bump into packets or frames that are a bit more complicated than this one, and may have more than one layer of information in them. Try to imagine how easy it would be to come up with the appropriate WHEN expression. It might get really ugly guys. But, and there's always a but, the solution is just a minute away.

Time to unveil the curtain - Progressive Coverage is all about avoiding WHEN commands, and instead - using multiple coverage events. So in our sample reference model we'll do this:

{code}

```
unit rx_ref_model {
 !curr_pkt: packet; // currently processed packet
 event cover_curr_pkt; // coverage event
 event cover_header; // coverage event
 event cover_payload; // coverage event
 // method port that receives a packet from somewhere
 receive_packet: in method_port of packet_type is instance;
 // implementation of the receiving method
 receive_packet(pkt: packet) is {
 curr_pkt = pkt; // copy pkt to be used locally
 if pkt.packet_valid then {
 process_header();
 emit cover_curr_pkt;
 };
 };
 process_header() is {
 // do something and then
 if pkt.header_crc_ok then {
 process_payload();
 emit cover_header;
 };
 };
 process_payload() is {
 // do something
 if pkt.payload_crc_ok then {
 // do something
 emit cover_payload;
 };
 };
 // now we'll add the dedicated coverage groups
 cover cover_curr_pkt is {
 // cover something, but not header nor payload
 };
 cover cover_header is {
 item header // don't need when
 };
 cover cover_payload is {
 item payload_size: uint = payload.size() // don't need when
 };
 };
```

{/code}

So we got rid of the WHENs, and created a sequence of coverage events, that either occur or not, according to the parsing progress - hence the term Progressive Coverage. The nice thing about it is that in large systems it really makes life easier to deal only with the coverage items that are relevant to the current phase. The alternative - complicated WHEN expressions - is not only harder to write, but also more error-prone so that you might end up with missing coverage buckets, or God forbid - false ones.