Doing the Impossible: Using Formal Verification on Packet Based Data Paths

Doug Smith
Doulos
Austin, Texas

***Abstract*-Formal verification is known to work well in areas like control logic, interface protocols, and so on, but it is often dismissed for use on data paths since capacity becomes a significant issue. In particular, packet based protocols have potentially very large state spaces, which can pose a problem for formal. However, in this paper, a step by step process is presented, showing how to decompose a frame of data into simple formal constraints, modeling code, and assertions, which allows formal to fully explore the entire packet state space.**

I. Introduction

All verification tools and flows have their limitations. Formal verification is known to be limited by its capacity due to *state space* caused by a large cycle depths and a large cones of influence. Thus, it is commonly believed that using formal on data paths typically does not work or at least requires abstraction. In general, this is true. Therefore, packet based protocols spanning multiple cycles and carrying large payloads would be virtually impossible, at least in theory. For example, an Ethernet TCP/IP packet can transfer over 9000+ bytes of data in a jumbo frame. It is easy to understand why most would consider this too large of a data path for formal verification to handle.

However, in the words of Alexander the Great, "There is nothing impossible to him who will try." Indeed, even with Ethernet jumbo frames, it is possible for formal verification to explore the entire Ethernet state space. The trick is understanding that while 9000+ bytes may be sent, the *valid state space* is actually quite small.

Consider the packet structure of an Ethernet jumbo frame illustrated in Figure [1]. Most of the jumbo frame is comprised of the data payload, which can have any value. For this part of the frame, we can let formal pick any random data it wants, removing the data portion from the state space needed to explore. What we are really interested in verifying with packet-based protocols is: (1) does the packet get parsed correctly? and (2) is the embedded control information in the packet correct? In other words, we are really verifying the control logic that is driven by the information from the packet.

With the payload removed from the formal state space, there are only a handful of fields remaining. While these fields may be quite large, most fields only have a few valid values, which we can easily constrain to reduce the state

		0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset												
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
24	192																																
28	224																																
32	256																																
36	260	Data																															
...	...																																
9036	72288																																

Figure 1: Potential IPv4 state space up to jumbo frames of 2^{72288} bit combinations [1].

space further. Likewise, we can break up the packet into smaller structures so that each field of the packet is handled separately instead of as one large state space. In other words, packet-based frames and protocols are not beyond the scope of formal verification; rather, formal can prove that your packet parser handles every packet correctly and finds all invalid packet corner cases[1], which is something hard to claim with other verification flows.

Using a small packet based protocol such as the CAN bus, this paper will outline a seven step procedure for modeling packet data for formal verification, which can be used for any packet-based protocol.

II. MODELING PACKET BASED INPUT

A. Model the control logic

The first step to sending a packet of data is defining the hand-shaking protocol between the producer and the consumer. This is typically handled using a start-of-packet (SOP) and end-of-packet (EOP) handshake protocol (see Figure 2). This can be modeled using helper code or formal constraints.

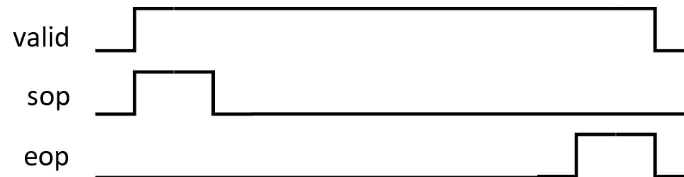

Figure 2: Packet handshake protocol.

The CAN bus protocol refers to a packet as a frame[2] and does not explicitly use a handshaking protocol. However, these handshaking signals can be used to constrain formal and are shown here for other protocols that use them. The control logic could be modeled with some procedural code as follows:

```
1   module canbus (input clk,
2                  input rst,
3                  input rx,
4                  output tx,
5                  input tx_rx);       // Transmitter or receiver
6
7      bit pkt_sof;      // Start of frame
8      bit pkt_eof;      // End of frame
9      bit pkt_vld;      // Valid packet
10
11
12     // ------------------------------------------------------------------
13     // (1) Model the control logic
14     // ------------------------------------------------------------------
15     bit       in_progress;               // Frame transaction in progress
16     bit [7:0] total_bits;                // Total frame bits
17     bit [7:0] tx_bits;                   // Number of frame bits transmitted
18
19     // Create an active flag
20     wire active = pkt_sof | in_progress;
21
22     always @(posedge clk or posedge rst) begin
23             if ( rst )     in_progress <= 1'b0;
24        else if ( pkt_eof ) in_progress <= 1'b0;
25        else if ( pkt_sof ) in_progress <= 1'b1;
26     end
27
```

Using this modeling code, formal constrains are easily specified to create the handshaking signals:

[1] Provided the valid values of the packets are fully specified.

[2] The terms packet and frame are used interchangeably in this paper.

```
28     default clocking cb @(posedge clk); endclocking
29
30     // Control signal constraints
31     property prop_transfer;
32        pkt_sof <-> pkt_vld[*1:$] ##0 pkt_eof;
33     endproperty
34
35     asm_pkt_vld: assume property ( active <-> pkt_vld );
36     asm_pkt_sof: assume property ( $rose(pkt_vld) |-> $rose(pkt_sof) );
37     asm_pkt_eof: assume property ( pkt_vld && (tx_bits >= total_bits) <-> pkt_eof );
38     asm_pkt_notsof: assume property ( in_progress |-> !pkt_sof );
```

The `prop_transfer` property specifies the waveform shown in Figure 2. The other assumptions constrain formal from toggling the valid, start-of-frame, and end-of-frame signals while the transfer is in progress. The `total_bits` and `tx_bits` signals will be assigned in a later step.

B. Define the packet structure

The next step is allocating a vector or array to hold the generated packet. While it may be tempting to define one large vector, the larger the vector, the larger the state space. By breaking the packet into smaller chunks, it makes it easier for formal to synthesize and reduces the number of constraints needed for each field in the packet. For this paper, we implement the CAN bus protocol as shown in Figure 3.

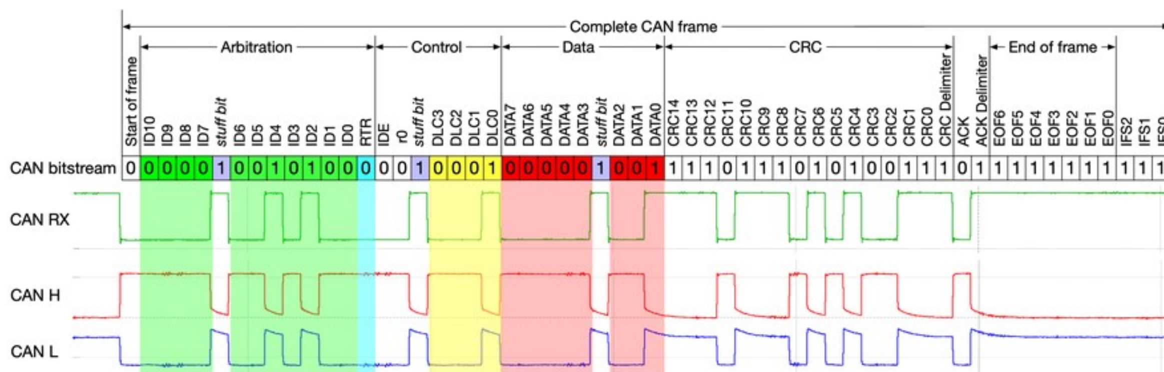

Figure 3: CAN bus protocol [2].

Examining the CAN frame, the largest field is the data field, which can be up to 8 bytes, and the CRC is the second largest field at 16 bits. Therefore, we could arbitrarily break the packet into either 8 bits or 16 bits chunks. The size of the chunk will determine the number of properties and potentially the speed of the formal analysis. For this example, we use 8 bits as our chunk size, and we use a standard CAN frame and not the extended frame format. (However, we do include bit stuffing, but bit stuffing is addressed when we transmit the data in another step).

The next step is to define an 8 bit structure for each part of the packet. For parts that are greater than 8 bits, we will define multiple structures to represent the field. Of course, some parts of the structure may have extra bits if a field is not a multiple of 8, but anything extra will simply become unused bits[3]. Here is one possible way to define structs for the CAN packet (note, since the data is transmitted from MSB to LSB, any unused bits will be included at the bottom of the structure):

```
39     // --------------------------------
40     // (2) Define the packet structure
41     // --------------------------------
42     // ------------------
43     // Start of frame
44     // ------------------
45     typedef struct packed {
46        bit       sof;
47        bit [6:0] unused;
48     } start_of_frame_t;
49
50
```

[3] The smaller the packet is sliced, the less bits go unused, but at the cost of more structures and coding. The state space is unaffected by the extra bits because they are not used and pose no performance issue for the formal tool.

```
51    // ------------------
52    // Arbitration
53    // ------------------
54    typedef struct packed {
55       bit [10:3] id;
56    } arbitration_high_t;
57
58    typedef struct packed {
59       bit [2:0] id;
60       bit       rtr;
61       bit [3:0] unused;
62    } arbitration_low_t;
63
64    // ------------------
65    // Control
66    // ------------------
67    typedef struct packed {
68       bit       ide;
69       bit       r0;
70       bit [3:0] dlc;
71       bit [1:0] unused;
72    } control_t;
73
74    // ------------------
75    // Data
76    // ------------------
77    typedef struct packed {
78       bit [7:0] value;
79    } data_t;
80
81    // ------------------
82    // CRC
83    // ------------------
84    typedef struct packed {
85       bit [14: 7] crc;
86    } crc_high_t;
87
88    typedef struct packed {
89       bit [6:0] crc;
90       bit       unused;
91    } crc_low_t;
92
93    // Break out the CRC delimiter
94    // so no bit stuffing occurs
95    // during the delimiter
96    typedef struct packed {
97       bit       crc_delimiter;
98       bit [6:0] unused;
99    } crc_delimiter_t;
100
101
102   // ------------------
103   // Acknowledge
104   // ------------------
105   typedef struct packed {
106      bit       ack;
107      bit       ack_delimiter;
108      bit [5:0] unused;
109   } ack_t;
110
111   // ------------------
112   // End of frame
113   // ------------------
114   typedef struct packed {
115      bit [6:0] eof;
116      bit       unused;
117   } end_of_frame_t;
118
119   typedef struct packed {
120      bit [2:0] ifs;
121      bit [4:0] unused;
122   } inter_frame_spacing_t;
```

With the packet structures defined, we can define a packed union to represent any part of the packet or frame. We include a member called `qbits` to explicitly access the packed union as a flatten queue of bits:

```
123   // Combined packet type
124   typedef union packed {
125      start_of_frame_t       sof;
126      arbitration_high_t     arb_h;
127      arbitration_low_t      arb_l;
128      control_t              ctrl;
129      data_t                 data;
130      crc_high_t             crc_h;
131      crc_low_t              crc_l;
132      crc_delimiter_t        crc_delimiter;
133      ack_t                  ack;
134      end_of_frame_t         eof;
135      inter_frame_spacing_t  ifs;
136      bit [7:0]              qbits;
137   } pkt_item_t;
```

With our packet items defined, we now create an array that contains our CAN bus frame. Since we have 11 structures and the data can be up to 8 bytes in a frame, we set the size of the array to be at least 18 plus an extra array element for terminating the frame:

```
138    // Packet items
139    localparam NUM_ITEMS = 1 +  // SOF
140                           1 +  // ARB_H
141                           1 +  // ARB_L
142                           1 +  // CTRL
143                           8 +  // DATA
144                           1 +  // CRC_H
145                           1 +  // CRC_L
146                           1 +  // CRC_DELIMITER
147                           1 +  // ACK
148                           1 +  // EOF
149                           1 +  // IFS
150                           1;   // NONE
151
152    pkt_item_t packet [NUM_ITEMS+1];
```

When we define the packet array, we allocate one more location so our formal properties will synthesize and work correctly, which is explained in the last step.

Associated with each chunk in the frame is another structure to keep track of its field kind or type, length, and a running tally of the overall length (as the number of bits or bytes depending on the protocol) of the frame. While this information can be embedded in each packet structure above, by separating out this information, we keep the size of the packet structure smaller for the formal tool and ensure that it does not affect the packet's state space. For example, the following code defines this additional structure:

```
153    // -----------------
154    // Packet Structure
155    // -----------------
156    typedef enum bit [3:0] { SOF, ARB_H, ARB_L, CTRL, DATA, CRC_H, CRC_L,
157                             CRC_DELIMITER, ACK, EOF, IFS, NONE } pkt_item_kind_t;
158
159    typedef struct packed {
160        pkt_item_kind_t  kind;
161        bit [7:0]        length;
162        bit [7:0]        total_length;
163    } packet_info_t;
164
165    packet_info_t packet_info [NUM_ITEMS+1];
```

Once again, the purpose of this step is to break the packet-based protocol into smaller, manageable chunks, which reduces the state space for formal and simplifies its formal synthesis. Conceptually, the arrays are illustrated in Figure 4, showing how the structures map back to the packet diagram. The `length` field represents the number of bits used in that element, and the `total_length` is the length of the entire packet/frame from that array element onwards.

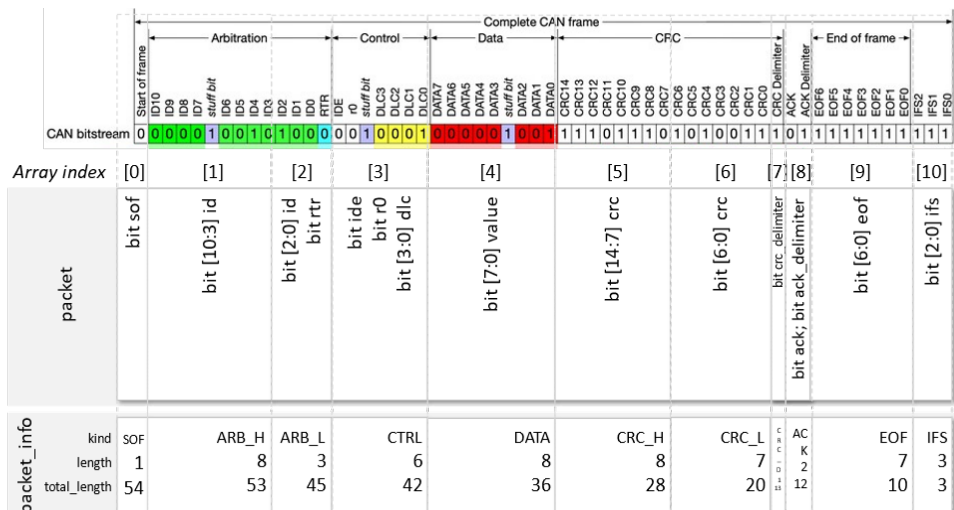

Figure 4: Example mapping of a CAN frame to the `packet` and the `packet_info` arrays.

C. *Define the packet constraints*

With the packet structures defined, we constrain our packet arrays to represent a valid, basic CAN frame. First, some helper constraints are added to keep track of each frame item, and tally up the total packet size:

```
166   // -------------------------------------------------------------
167   // (3) Define packet constraints
168   // -------------------------------------------------------------
169   sequence seq_kind(n, k);
170      packet_info[n].kind == k;
171   endsequence
172
173   sequence seq_total_length(n);
174      packet_info[n].total_length == packet_info[n+1].total_length +
175                                      packet_info[n].length;
176   endsequence
177
178   sequence seq_length(n, l);
179      ( packet_info[n].length == l ) and seq_total_length(n);
180   endsequence
181
182   sequence seq_terminate_length(n);
183      ( packet_info[n].length == 0 ) &&
184      ( packet_info[n].total_length == 0 );
185   endsequence
```

The `n` passed into the named sequences represents the element in the packet arrays shown above. With the `seq_total_length` sequence, the total length is calculated using the current element's length and the next element's length (`n+1`). The `seq_length` sequence sets the length and then calls the `seq_total_length`. This technique of reaching into the previous or next element in the frame is one way to pass information about the entire packet to the header fields (like the total packet/frame length) or for error checking functions (like checksum or CRC).

Now a property constraint needs written for each element in the packet to specify its legal values. Each property is passed an index into the packet array (`n`), and the packet element's kind, length, and legal values for its structure's members are described within the named property:

```
186   //
187   // Start of frame
188   //
189   property prop_sof(n);
190      seq_kind(n,SOF) and
191      seq_length(n,1) and
192      (packet[n].sof.sof == '0) and
193      (packet[n].sof.unused == '0);
194   endproperty
195
196   //
197   // Arbitration
198   //
199   enum bit { DATA_FRAME   = 0,
200              REMOTE_FRAME = 1
201            } frame_type;
202   bit [10:0] id;
203
204   property prop_arb_h(n);
205      seq_kind(n,ARB_H) and
206      seq_length(n,8) and
207     (packet[n].arb_h.id == id[10:3]);
208   endproperty
209
210   property prop_arb_l(n);
211      seq_kind(n,ARB_L) and
212      seq_length(n,4) and
213      (packet[n].arb_l.id == id[2:0])
214      and
215   (packet[n].arb_l.rtr == frame_type)
216      and
217      (packet[n].arb_h.unused == '0);
218   endproperty
219
220   //
221   // Control
222   //
223   bit [3:0] payload_size;
224   property prop_control(n);
225      seq_kind(n,CTRL) and
226      seq_length(n,6) and
227      (packet[n].ctrl.ide == 0) and
228      (packet[n].ctrl.r0  == 0) and
229  (packet[n].ctrl.dlc == payload_size)
230      and
231      (packet[n].ctrl.unused == '0);
232   endproperty
233
234   //
235   // Data payload
236   //
237   bit [7:0] random_data;
238
239   property prop_data(n);
240      seq_kind(n,DATA) and
241      seq_length(n,8) and
```

```
242 (packet[n].data.value == random_data);
243   endproperty
244
245   //
246   // CRC
247   //
248   bit [15:0] crc;
249
250   property prop_crc_h(n);
251      seq_kind(n,CRC_H) and
252      seq_length(n,8) and
253    (packet[n].crc_h.crc == crc[14:7]);
254   endproperty
255
256   property prop_crc_l(n);
257      seq_kind(n,CRC_L) and
258      seq_length(n,7) and
259     (packet[n].crc_l.crc == crc[6:0]);
260   endproperty
261
262   property prop_crc_delimiter(n);
263      seq_kind(n,CRC_DELIMITER) and
264      seq_length(n,1) and
265  packet[n].crc_delimiter.crc_delimiter
    == 1);
266   endproperty
267
268   //
269   // ACK
270   //
271   property prop_ack(n);
272      seq_kind(n,ACK) and
273      seq_length(n,2) and
274      (packet[n].ack.ack == tx_rx) and
275  (packet[n].ack.ack_delimiter == '1);
276   endproperty
277
278   //
279   // End of frame
280   //
281   property prop_eof(n);
282      seq_kind(n,EOF) and
283      seq_length(n,7) and
284      (packet[n].eof.eof == '1);
285   endproperty
286
287   //
288   // Inter-frame spacing
289   //
290   property prop_ifs(n);
291      seq_kind(n,IFS) and
292      seq_length(n,3) and
293      (packet[n].ifs.ifs == '1);
294   endproperty
295
296   //
297   // Terminal for packet
298   //
299   property prop_none(n);
300      seq_kind(n,NONE) and
301      seq_terminate_length(n);
302   endproperty
```

To better understand these properties, consider the start-of-frame:

```
189     property prop_sof(n);
190        seq_kind(n,SOF) and
191        seq_length(n,1) and
192        (packet[n].sof.sof == '0) and
193        (packet[n].sof.unused == '0);
194     endproperty
```

On line 190, `seq_kind` is called with the array index and the `kind` as defined by the `pkt_item_kind_t` enumeration (line 157). This sets the `packet_info[n].kind` element to the start-of-frame or `SOF`, and `seq_length` sets `packet_info[n].length` to 1, adds 1 to the total frame bit count, and assigns it to `packet_info[n].total_length`. These two sequences are included in all the properties, and then any individual field constraints required by the bus protocol are included. For example, the `sof` field is set to 0 per the CAN bus protocol. For purposes of the CRC calculation, the `unused` bits are also constrained to 0 since zeros are passed over in the CRC calculation.

Notice that the packet's payload, which can be up to 8 bytes in a basic CAN bus frame, is specified using only one named property, `prop_data`. This property will be applied multiple times for each byte in the payload. Since we wish to transfer just random data, we are using an unconstrained formal control point, `random_data`, and assigning it for our data payload. For calculating the CRC, we define a local variable called `crc`, which will be assigned the calculated CRC value for the CAN frame. The CRC calculation is performed using a function, which is shown in the last step. By themselves, these named properties do nothing, but in the next step, we apply these properties to each element of the packet array so formal knows how to generate the valid CAN frame.

D. Apply the packet constraints

The next step is to apply the property constraints defined above. This is where the actual packet or frame is defined. When we verify our design, we want to check that both the valid and invalid packets are handled correctly;

however, it is easier to define the valid packets since they are generally fully specified. For now, our focus will be on defining valid packets, but later we will show how to generate illegal packets.

The key to creating a packet or frame is defining a top level property with conditional statements[4]. Here is what our CAN frame property looks like:

```
303    // ---------------------------------------------------------------
304    // (4) Apply packet constraints
305    // ---------------------------------------------------------------
306    property prop_pkt(n);
307            if ( packet_info[n].kind == SOF           ) prop_arb_h(n+1)
308       else if ( packet_info[n].kind == ARB_H         ) prop_arb_l(n+1)
309       else if ( packet_info[n].kind == ARB_L         ) prop_control(n+1)
310       else if ( ( payload_size == 0    ) && ( packet_info[n].kind == CTRL) ) prop_crc_h(n+1)
311       else if ( ( payload_size  > 0    ) && ( packet_info[n].kind == CTRL) ) prop_data(n+1)
312       else if ( (( payload_size+3) >  n) && ( packet_info[n].kind == DATA )) prop_data(n+1)
313       else if ( (( payload_size+3) <= n) && ( packet_info[n].kind == DATA )) prop_crc_h(n+1)
314       else if ( packet_info[n].kind == CRC_H         ) prop_crc_l(n+1)
315       else if ( packet_info[n].kind == CRC_L         ) prop_crc_delimiter(n+1)
316       else if ( packet_info[n].kind == CRC_DELIMITER ) prop_ack(n+1)
317       else if ( packet_info[n].kind == ACK           ) prop_eof(n+1)
318       else if ( packet_info[n].kind == EOF           ) prop_ifs(n+1)
319       else                                             prop_none(n+1);
320    endproperty
```

The first element in the `packet_info` array needs set and then all the other constraints fall into place. The code for that is shown in a later step, but Figure 5 illustrates how the `prop_pkt` property is applied to the `packet_info` array, and how the *n+1* index actually sets the element type for the *next element* in the array. The `prop_pkt` property is applied to all packet elements, which is shown in the final step.

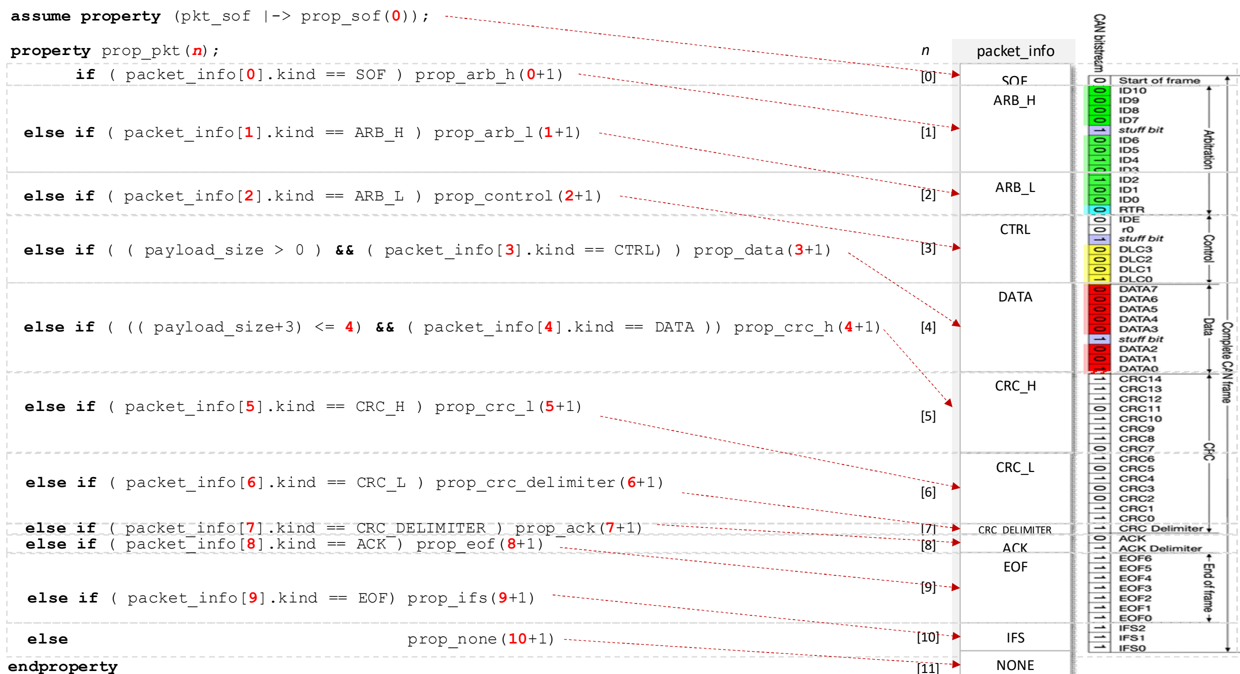

Figure 5: How the `prop_pkt` property sets each element in the `packet_info` array.

E. Model the packet driving logic

Sending the packet is easily accomplished with some synthesizable SystemVerilog helper code. The helper code also handles the bit stuffing used by the CAN bus protocol. Bit stuffing is used to maintain synchronization by

[4] SVA allows a *case* statement within a property, but not all formal tools currently support it.

inserting a bit of the opposite polarity every time 5 consecutive bits of the same polarity are transmitted. It is slightly more complicated because bit stuffing is not used while transmitting the fixed length part of the frame from the CRC delimiter to the inter-frame spacing [2]. Plus, the bit stuffing is not used in the calculation of the CRC.

The CAN bus sends one bit of data at a time. Stepping through the CAN frame is accomplished using two pointers—one that specifies the unpacked dimensions of the `packet` array and one that steps through the packed dimensions of each array element. The unpacked dimension pointer is `p` in the following code, and `n` represents the packed dimensions pointer. Instead of driving directly onto the `tx` output port, an intermediate signal `tx_out` is used, and then assigned to the `tx` output using a formal constraint in the next step. The reason for this is to simplify the helper code modeling and give the flexibility of controlling the output easily with additional formal constraints. A transmitted bit counter, `tx_bits`, is maintained for determining the end-of-frame as shown above on line 17 of the example code. For completeness, the bit stuffing code is included, but it is slightly greyed out to focus on the packet driving logic.

```
321    // ------------------------------------------------------------------
322    // (5) Model the packet driving logic
323    // ------------------------------------------------------------------
324    bit       tx_out;             // Data to drive to the output
325    bit [3:0] p;
326    bit [2:0] n;
327    bit       prev_tx;            // Extra bit stuffing logic in grey
328    bit [2:0] same;
329
330    always @(posedge clk or posedge rst)
331    begin
332       if ( rst ) begin
333          tx_out  <= '0;
334          p       <= '0;
335          n       <= $bits(pkt_item_t) - 1'b1;
336          tx_bits <= '0;
337          prev_tx <= '0;
338          same    <= '0;
339       end
340       else begin
341
342       // -------------------
343       // Transfer the data
344       // -------------------
345       if ( pkt_vld ) begin
346
347          //
348          // Insert bit stuffing
349          //
350          if (( packet_info[p].kind inside { [SOF:CRC_L] } ) && ( same == 5 )) begin
351
352             tx_out  <=  ~prev_tx;
353             same    <= '0;
354             prev_tx <=  ~prev_tx;
355
356          end
357          else begin
358
359             // Drive the packet data
360             tx_out <= packet[p].qbits[n];
361
362             // Keep track of how many bits sent
363             tx_bits <= tx_bits + 1'b1;
364
365             // Update pointers
366             if ( ( $bits(pkt_item_t) - n ) == packet_info[p].length ) begin
367                if ( packet_info[p].kind == NONE )
368                   p <= '0;                    // Wrap back to beginning
```

```
369                 else
370                     p <= p + 1'b1;               // Next packet item
371
372                 n <= $bits(pkt_item_t) - 1'b1; // Start with the top bit
373              end
374              else begin
375
376                 // Next bit in the packet item
377                 n <= n - 1'b1;
378              end
379
380              // Bit stuffing tracking
381              if ( packet[p].qbits[n] == prev_tx ) begin
382                 same    <= same + 1'b1;
383              end
384              else begin
385                 same    <= '0;
386                 prev_tx <= packet[p].qbits[n];
387              end
388            end
389          end
390          else begin
391
392             // Clear when not valid
393             p       <= '0;
394             n       <= $bits(pkt_item_t) - 1'b1;
395             prev_tx <= '0;
396             same    <= '0;
397          end
398       end
399    end
```

The packet driving logic is highlighted on line 360. The pointers `p` and `n` are incremented through the packet, and unused bits are skipped over by looking at the length of each array element as defined in the `packet_info` array (line 366). Formal synthesizes this code into the logic shown in Figure 6.

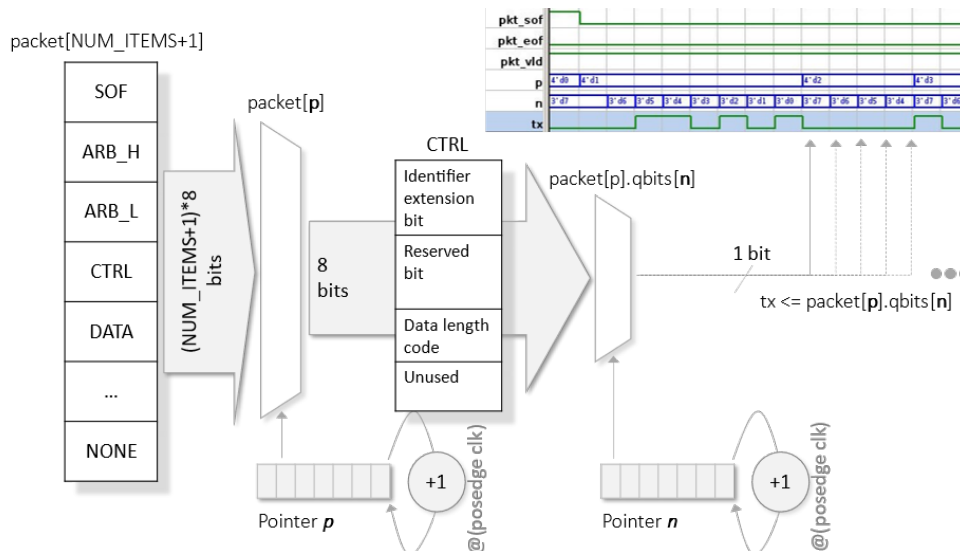

Figure 6: Diagram of the modeling code used to drive the packet data.

For the bit stuffing logic in grey, the data transmitted is saved in `prev_tx` and a counter called `same` keeps track of the number of consecutive occurrences of the same polarity. When the counter reaches 5, data of the opposite polarity is driven onto the output except during the fixed length parts at the end of the frame.

F. Generate the packet

With everything in place, the last and final step is to add the formal constraints to create the packet of data. First, the packet needs to be initialized:

```
400    // ----------------------------------------------------------------
401    // (6) Generate the packet
402    // ----------------------------------------------------------------
403
404    // Initialize the packet
405    asm_pkt_init_start   : assume property ( pkt_sof |-> prop_sof(0) );
406    asm_pkt_init_last    : assume property ( prop_none(NUM_ITEMS) );
```

Constraint `asm_pkt_init_start` assigns the first element in the packet to be a start-of-frame. The last element in the packet array is assigned to be of type `NONE`, meaning that it is an unused element. The reason for including the extra element at the end of the packet array (`NUM_ITEMS+1`) is because the packet constraints use *n+1*. Since the properties need to be synthesizable, the extra element is added so that no index out-of-bounds error is generated when referencing `packet[n+1]` or `packet_info[n+1]`. In the above code, `prop_none(NUM_ITEMS)` initializes that last element to `NONE` since it is not used.

Next, each element in the packet is constrained using the `prop_pkt` property discussed previously:

```
407    // Create the packet/frame
408    for (genvar i = 0; i < NUM_ITEMS-1; i++ )
409    begin : pkt_init
410       asm_init_pkt : assume property ( pkt_sof |-> prop_pkt(i) );
411    end
412
413    asm_set_total_frame_length :
414                     assume property ( total_bits == packet_info[0].total_length );
415
416    asm_payload_size       : assume property ( payload_size inside { [0:8] } );
417    asm_payload_size_stable: assume property ( pkt_vld |-> $stable(payload_size) );
```

The generate block handles the packet initialization by assigning the `prop_pkt` property constraint to each element. Recall, the `total_length` bit count represents the total number of bits in the frame from that position onwards. Therefore, the total length can be found in element 0 of the `packet_info` array as shown on line 414. The `total_bits` variable is set with this assumption and then used to control the end-of-frame signal when the number of transmitted bits (`tx_bits`) reaches `total_bits`. In order to constrain the payload size (i.e., number of data elements), the payload is constrained on line 416.

While the `asm_init_pkt` constraint initializes most of the frame, the CRC still needs to be calculated. A synthesizable function is defined which incrementally calculates the CRC using one byte at a time. While the specific implementation is not important, it is included here for reference as an example for calculating CRC or checksum:

```
418    // ----------------------------------------------------------------
419    // CRC functions
420    // The following is taken from http://blog.qartis.com/can-bus
421    // ----------------------------------------------------------------
422    function bit [15:0] can_crc_next(bit [15:0] crc, bit [7:0] data);
423       crc ^= 16'(data) << 7;
424
425       for ( int i = 0; i < 8; i++ ) begin
426           crc <<= 1;
427           if ( crc & 16'h8000 ) begin
428              crc ^= 16'hc599;
429           end
430       end
431       return (crc & 16'h7fff);
432    endfunction : can_crc_next
433
```

```
434   function bit [15:0] calc_crc();
435       calc_crc = '0;
436
437       calc_crc = can_crc_next( calc_crc, { '0, packet[1].qbits[7:6] });
438       calc_crc = can_crc_next( calc_crc, { packet[1].qbits[5:4],
439                                            packet[2].qbits[7:2] });
440       calc_crc = can_crc_next( calc_crc, { packet[2].qbits[1:0],
441                                            packet[3].qbits[5:0] });
442
443       for ( int i = 4; i < 12; i++ ) begin
444           if (( payload_size + 4 ) > i ) begin
445              calc_crc = can_crc_next( calc_crc, packet[i].qbits );
446           end
447       end
448   endfunction : calc_crc
449
```

Recall on line 248, a variable named `crc` was defined and used within the named properties `prop_crc_h` and `prop_crc_l`. This variable can now be set using the `calc_crc()` function:

```
450   // Generate the CRC
451   asm_calc_crc: assume property ( pkt_vld |-> crc == calc_crc() && $stable(crc) );
452
```

With the CRC constrained, the frame is now complete. A formal constraint is used to assign the `tx_out` variable to the `tx` output, and the packet needs to be held stable during the packet transfer or formal will continue to change the packet structure each clock cycle:

```
453   // Drive the frame
454   asm_drive_data      : assume property (  pkt_vld |-> tx == tx_out );
455   asm_undriven        : assume property ( !pkt_vld |-> tx == 1'b1 );
456   asm_pkt_stable      : assume property (  pkt_vld |-> $stable(packet) );
457   asm_pkt_info_stable: assume property (  pkt_vld |-> $stable(packet_info) );
458
```

The `asm_drive_data` constraint performs the actual driving of the data to the output. The CAN bus protocol specifies a value of 1 when not driving so `asm_undriven` provides that functionality.

G. Check the packets

The last and final step is to write the actual formal assertion or cover properties to perform the formal verification. If driving into a packet parser, the design would typically include a status signal to indicate whether a packet has been successfully received and parsed. In that case, a typical assertion would be to assert that the status signal never indicates an error since we have defined only valid, legal packets. For this CAN bus example, we define a cover property to generate a waveform of the frame:

```
459   // Generate waveform of frame
460   cov_gen_packet:  cover property ( prop_transfer );
```

The cover property waveform is shown in Figure 7. The `packet_info` shows how the formal constraints have built the packet, specifically the type or `kind` of each element, which is used for constraining the packet elements. The union member `qbits` is used to drive the value onto the `tx` output, which is highlighted in blue in the waveform.

One modification to consider is the ability of specifying bad packets or frames for testing the design's handling of bad input. We start by defining a flag that specifies if the packet is valid (good) or invalid (bad):

```
461  // Used by formal tool to select good or bad packets
462  enum bit  { FALSE = 0, TRUE = 1 } pkt_good;
```

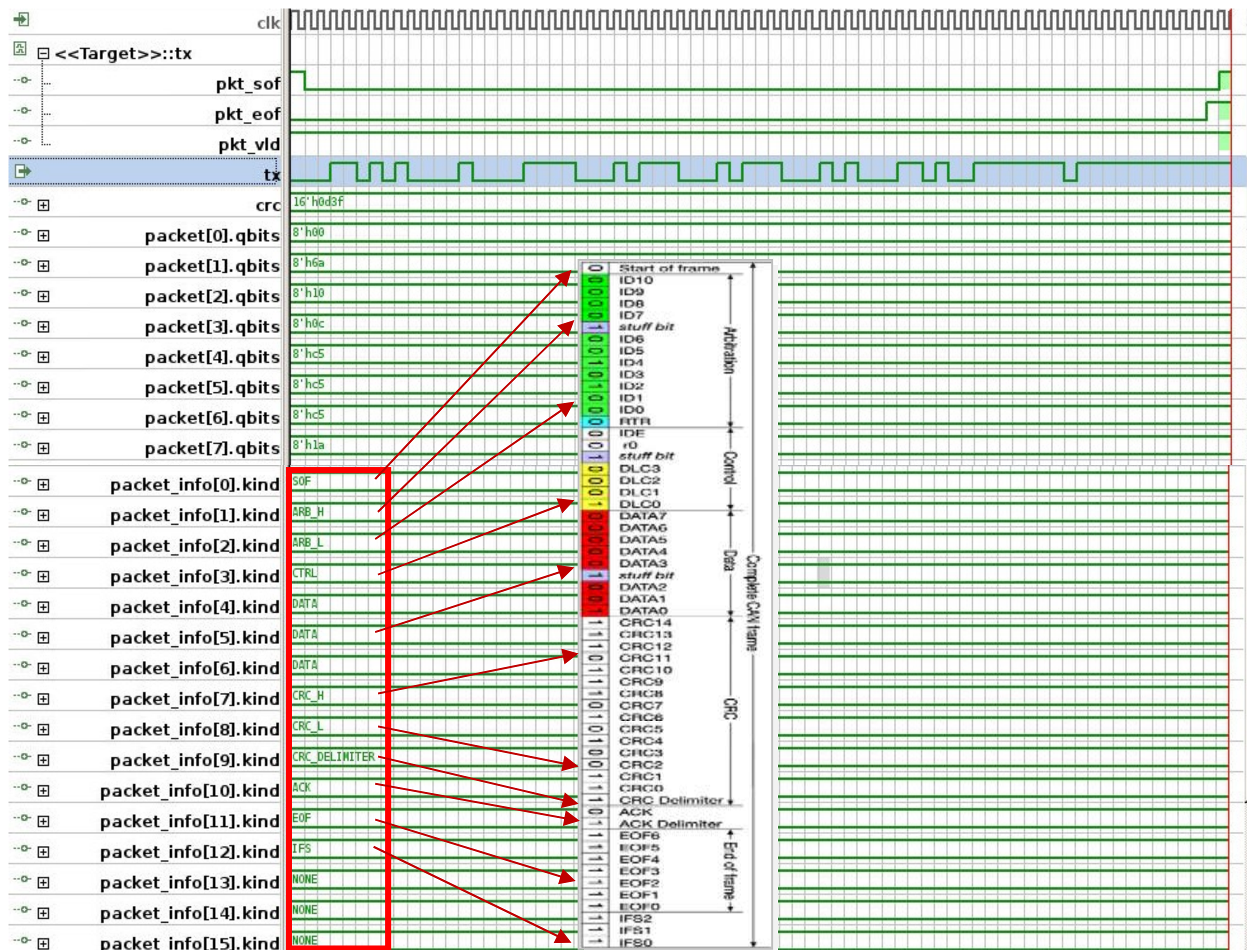

Figure 7: Example CAN frame generated by formal using a cover property.

Next, we modify the properties we already defined, creating a new property that will set the kind, length, and call our previous properties. For example,

```
189   property prop_sof(n);
190     seq_kind(n,SOF) and
191     seq_length(n,1) and
192     (packet[n].sof.sof == '0) and
193     (packet[n].sof.unused == '0);
194   endproperty
```

Is changed to:

```
189   property prop_sof(n);
190     (packet[n].sof.sof == '0) and
191     (packet[n].sof.unused == '0);
192   endproperty
193
194   // Add new property
195   property prop_pkt_sof(n);
195      seq_kind(n,SOF) and
196      seq_length(n,1) and
197      if ( pkt_good )     prop_sof(n)
198      else            not(prop_sof(n));
199   endproperty
```

The pkt_good flag tells formal to use the valid packet constraints or set it to the opposite. Once we modify each constraint, we modify the pkt_prop constraint to call the new properties instead (e.g., prop_pkt_sof shown above). Then to generate a good or bad packet, we just include the prop_good flag in the assertions or cover property:

```
463    cov_gen_good_packet : cover property (     pkt_good  and prop_transfer );
464    cov_gen_bad__packet : cover property ( not(pkt_good) and prop_transfer );
```

The **not**(pkt_good) causes formal to set pkt_good **==** FALSE, which in turn affects the conditional statements in the packet properties. With pkt_good set to false, only invalid packet data will be generated.

In this example, only cover properties have been shown, but this could easily be applied to assertions and checking the design. For example, we could assert that when a packet or frame is bad (i.e., **not**(pkt_good)), the packet is caught by the parser and marked as malformed:

```
465    // Define a sequence to indicate the design is parsing the packet
466    property pkt_parsing;
467       !parse[*0:$] ##1 parse[*1:$] ##1 !parse;
468    endproperty
469
470    ast_bad_pkt_marked_malformed :
471       assert property ( not(pkt_good) && pkt_sof && pkt_valid |->
472                                                malformed_signal within ( pkt_parsing ));
473
```

III. RESULTS

Performance between formal tools varies using this approach. Compilation is typically very quick (usually seconds) provided the packet structures are sliced small enough to be efficient for formal. The author has personally used this approach for verifying Ethernet jumbo frames (9000 bytes) on a real-world Ethernet parser. Generating a packet waveform as shown in Figure 7 takes a bit more time. In this example of a CAN bus frame, results vary between the major EDA vendors between 20 seconds to 1 minute. However, for the larger Ethernet example, generation times were typically between 3 to 6 minutes, but formal analysis could take a bit longer for proofs.

Developing the constraints and modeling code discussed in this paper is typically much less time than developing an equivalent UVM environment. For example, this CAN bus example was developed over a couple days, including learning the bus protocol. With this approach, no test cases were needed nor did any corner case need specified. All that is needed is the requirements for the design and the appropriate assertions and cover properties specifying whether to use a good or bad packet. Formal will do the rest and thoroughly test your design without weeks or even months of developing a simulation testbench. The added advantage is that the formal constraints used to specify the packet or frame can also be used with simulation and complement other verification environments. Assumptions specified for formal become assertions in simulations so they can be used as extra checks in simulation to verify that the stimulus was generated correctly.

Indeed, the formal constraints are perhaps the easiest part to this approach. Developing the appropriate modeling code is typically much more time-consuming. In this example, adding in the bit stuffing and the CRC calculation proved to be the more difficult and time-consuming component than developing the constraints.

IV. SUMMARY

In summary, modeling a packet-based protocol for formal technology can be accomplished using the following seven steps:

1. *Model the control logic* – this refers to the logic that indicates a valid packet is available, usually including a start of packet, end of packet, and valid signal.
2. *Define the packet structure*s – break the packet into smaller chunks to reduce the state space. Find a small divisor to break the packets into a manageable size and make all the packet structures that size. If your packet interface transfers 8, 16, or 32 bits at a time, it might be easiest to match the interface bus width.
3. *Define the packet constraints* – the packet constraints define what are the valid values for each field in the packet.

4. *Apply the packet constraints* – applying the constraints involves a top-level conditional named property that can apply the constraints to the packet fields based on other fields in the packet, mode bits, or inputs.
5. *Model the packet driving logic* – this involves writing synthesizable code that drives the packet structures onto the design's packet interface.
6. *Generate the packet* – generating the packet is accomplished by applying the constraints with assume properties. Generate statements can be a great help when applying packet constraints.
7. *Check the packets* – lastly, create assertions to check for the correct design behavior when either valid or invalid packets are generated.

While not all data path problems are solvable using formal, this approach demonstrates that with a little ingenuity, a proven methodology, and understanding how a formal tool synthesizes and solves problems, even complicated packet-based protocols are within the possibility of formal verification.

REFERENCES

[1] Wikipedia, "IPv4," [Online]. Available: https://en.wikipedia.org/wiki/IPv4. [Accessed 26 07 2022].

[2] Wikipedia, "CAN bus," [Online]. Available: https://en.wikipedia.org/wiki/CAN_bus. [Accessed 28 July 2022].