Decoder and Encoder for Dynamic C-Structures

Timon Lapawczyk

April 20, 2015

1 Introduction

The Decoder and Encoder functions for all static packets now make use of the Bitstream_Write and Bitstream_Read functions. Therefore one does not have to calculate the bitpositions himself when it comes to the C-Code.

However this does not apply to the formal specification. While checking a predicate like EqualBits, frama-c can not access the states within the function, to obtain the current bitpos.

```
predicate EqualBits(Bitstream* stream, integer pos, Adhesion_Factor* p) =
  EqualBits(stream, pos,      pos + 8,   p->NID_PACKET)       &&
  EqualBits(stream, pos + 8,  pos + 10,  p->Q_DIR)            &&
  EqualBits(stream, pos + 10, pos + 23,  p->L_PACKET)         &&
  EqualBits(stream, pos + 23, pos + 25,  p->Q_SCALE)          &&
  EqualBits(stream, pos + 25, pos + 40,  p->D_ADHESION)       &&
  EqualBits(stream, pos + 40, pos + 55,  p->L_ADHESION)       &&
  EqualBits(stream, pos + 55, pos + 56,  p->M_ADHESION);
```

In the next step we focus on the C-Code and leave the formal specification aside. Hence we can for now ignore the problem with the predicates and focus on other problems.

2 Bitstream_Write and Bitstream_Read

With the switch to the Bitstream functions the current bitpos is saved within the Bitstream structure and rewritten during the Bitstream_Read and Bitstream_Write calls.

That makes the Decoder and Encoder functions a lot cleaner.

In the example below all Read calls are independent from one another.

```
int Infill\_location\_reference\_Decoder(Bitstream* stream, Infill\_location\
    _reference* p)
{
    if (NormalBitstream(stream, INFILL_LOCATION_REFERENCE_BITSIZE))
    {
        uint8_t* addr = stream->addr;
        const uint32_t size = stream->size;
        const uint32_t pos = stream->bitpos;

        p->NID_PACKET   = Bitstream_Read(stream, 8);
```

```
        p->Q_DIR         = Bitstream_Read(stream, 2);
        p->L_PACKET      = Bitstream_Read(stream, 13);
        p->Q_NEWCOUNTRY  = Bitstream_Read(stream, 1);
        p->NID_C         = Bitstream_Read(stream, 10);
        p->NID_BG        = Bitstream_Read(stream, 14);

        return 1;
    }
    else
    {
        return 0;
    }
}
```

3 Optional variables

As it happens, the example above contains an optional variable.

- The value for NID_C is only read if Q_NEWCOUNTRY has the value 1.
- The dependency on the value of Q_NEWCOUNTRY can be realized via an if clause.
- Since all Bitstream_Read calls are indpendent the bitpos for NID_BG remains unchanged if NID_C is not read and adds up automatically if NID_C is read.

The NID_C line from above is changed into

```
if (p->Q_NEWCOUNTRY == 1)
{
    p->NID_C        = Bitstream_Read(stream, 10);
}
```

The first question at hand is how do we calculate the value for INFILL_LOCATION_REFERENCE_BITSIZE to check NormalBitstream?

Until now this value has been set in the Infill_location_reference.h file, as the sum of the bitsizes of all variables. Now the actual bitsize may depend on one or more values, read during runtime. There is no access to these values when deciding, whether the stream is long enough to read the whole packet from.

To maintain some error handling we do not change this value. We can still assure to capture all error cases, where there are not enough bits to read from, in the stream. However we can not assure anymore that a return of 0 means that there were not enough bits. A not read optional variable might have also been the cause of the error.

4 Bitsize in Decoder Branch

Maybe we can recycle some of our thoughts on the header files. The L_PACKET value equals the number of bits, that are actually transfered and therefore the neccessary length of the stream. If we read the L_PACKET bits together with the NID˙PACKET value in the decoder branch, we can hand it on to the decoder.