OETCS/WP4/D4.2.2

openETCS

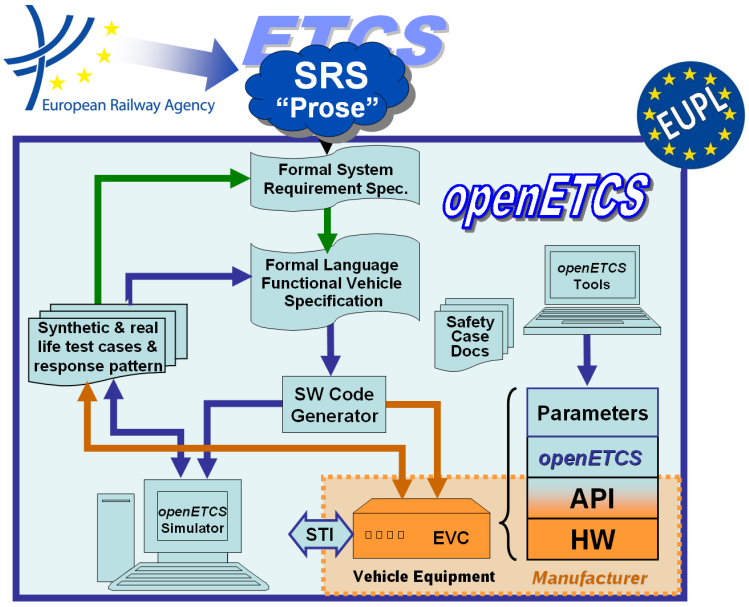
ITEA2 Project
Call 6 11025
2012 – 2015

Work Package 4: “Validation & Verification Strategy”

First Validation and Verification Report on Implementation/Code

Marc Behrens and Jens Gerlach

November 2013



Funded by:

Federal Ministry of Education and Research

Région de Bruxelles-Capitale

This page is intentionally left blank

Work Package 4: "Validation & Verification Strategy"

OETCS/WP4/D4.2.2
November 2013

First Validation and Verification Report on Implementation/Code

Marc Behrens

WP4 Leader

Jens Gerlach

WP4.3 Task Leader (Validation and Verification of Implementation/Code)

Description of work

Prepared for openETCS@ITEA2 Project

Abstract: This work package will comprise the activities concerned with verification and validation within openETCS. This includes verification & validation of development artifacts, that is, showing that models and code produced correctly express or implement what they are supposed to. And also, methods and tools to perform such tasks will be evaluated with the goal of assembling a suitable method and tool chain to support a full development.

Table of Contents

1 Introduction 5
2 Formal Verification of Bitwalker 5
2.1 Verification Objectives 5
2.2 The Function `Bitwalker_Peek` 5
2.3 The Function `Bitwalker_Poke` 9
2.4 Interaction of `Bitwalker_Peek` and `Bitwalker_Poke` 13
2.5 Open Issues 13
3 SQS 13
4 CEA LIST 13
5 Systerel 13
6 Conlusions 13

Figures and Tables

Figures

Figure 1. Array indices and bit indices in a bit stream 5
Figure 2. A bit sequence within a bit stream 6

Tables

1 Introduction

2 Formal Verification of Bitwalker

To be done

2.1 Verification Objectives

To be done

2.1.1 Functionality

To be done

2.1.2 Robustness

To be done

2.2 The Function `Bitwalker_Peek`

To be done

2.2.1 Informal Specification

We introduce some auxiliary concepts and formulate general assumptions:

- A *bit stream* is an array containing elements of type `uint8_t`.

  A bit stream of length n contains $8n$ bits.

- A bit stream is *valid* if the array is valid.

- A bit stream can be indexed both by its array indices and its *bit indices*.

  Figure 1 shows the difference between array indices and bit indices in a bit stream. The two bit indices, 0 and 14, mark bit positions in the first and second array element, respectively.

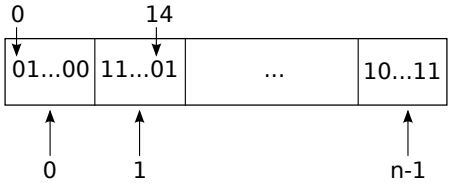


Figure 1. Array indices and bit indices in a bit stream

- A *bit sequence* is a consecutive sequence of bits within a bit stream as represented in Figure 2.

  A bit sequence is given by the position of its first bit (a bit index in the bit stream) and its *length*, that is, the number of bits it contains.

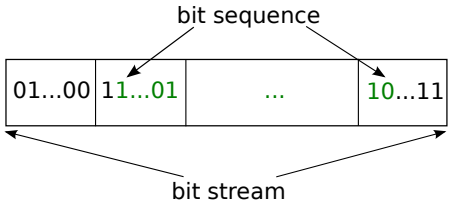


Figure 2. A bit sequence within a bit stream

- A bit sequence of length l that starts at bit index p is *valid* with respect to a bit stream of length n if the following conditions are satisfied

$$0 \leq p \leq 8n$$
$$0 \leq p + l \leq 8n$$

- We assume that the C-types `unsigned int` and `int` have a width of 32 bits.

Now we specify `Bitwalker_Peek` with the introduced auxiliary concepts. The function `Bitwalker_Peek` reads a bit sequence from a bit stream and converts it to an integer.

Its function signature reads as follows:

```
uint64_t  Bitwalker_Peek(unsigned int Startposition,
                         unsigned int Length,
                         uint8_t Bitstream[],
                         unsigned int BitstreamSizeInBytes);
```

The arguments have the following purpose:

- `Startposition` is the bit index in the bit stream where the bit sequence starts.
- `Length` is the length of the bit sequence.
- `Bitstream` is the array which provides the bit stream.
- `BitstreamSizeInBytes` is the length of the array containing the bit stream.

The following preconditions shall hold for the function arguments:

- `Bitstream` is a valid array of length `BitstreamSizeInBytes`
- `Length` $\leq$ `64` and
- `Startposition + Length` $\leq$ `UINT_MAX`.

Note that additional constraints are implicitly expressed by the use of *unsigned* integer types.

We continue with a more precise description of the desired behavior of `Bitwalker_Peek`. As mentioned, the function `Bitwalker_Peek` reads a bit sequence from a bit stream and converts it to a 64-bit unsigned integer.

The left most bit of the bit sequence is interpreted as the most significant bit. Thus, for a bit sequence $(b_0, b_1, \ldots, b_{n-1})$ the function returns the sum

$$b_0 \cdot 2^{n-1} + b_1 \cdot 2^{n-2} + \ldots + b_{n-1} \cdot 2^0 = \sum_{i=0}^{n-1} b_i \cdot 2^{(n-1)-i} \quad (1)$$

If the bit sequence is not valid, then the function returns `0`. This increases the robustness of the function.

2.2.2 Implementation

Listing 2.1 shows the C implementation of `Bitwalker_Peek` for which we aim to verify that it fulfills the informal specification. The case where the bit sequence is not valid is handled by the `if`-statement. For a valid sequence the summation of the bits is done in the `for`-loop. The array `BitwalkerBitMaskTable` is a `const` helper array to select a single bit in the `Bitstream`.

```
uint64_t Bitwalker_Peek (unsigned int Startposition, unsigned int
    Length,
                         uint8_t Bitstream[], unsigned int
                             BitstreamSizeInBytes)
{
    if (((Startposition + Length - 1) >> 3) >= BitstreamSizeInBytes)
        return 0;

    uint64_t retval = 0;

    unsigned int i;
    for (i = Startposition; i < Startposition + Length; i++)
    {
        uint8_t CurrentValue = Bitstream[i >> 3] &
            BitwalkerBitMaskTable[i & 0x07];

        retval = (retval << 1) + (uint8_t)(CurrentValue != 0);
    }

    return retval;
}
```

Listing 2.1. Implementation of `Bitwalker_Peek`

The implementation uses a great amount of bit operations which is quite a challenge for the formal verification. We will discuss this further in section 2.5.

2.2.3 Formal Specification with ACSL

In order to verify that the given implementation of `Bitwalker_Peek` fulfills the informal specification, we have to formalize the specification. Listing 2.2 shows such a formalization in ACSL for `Bitwalker_Peek`.

We specify a function contract for `Bitwalker_Peek` containing preconditions and postconditions introduced by the key words **`requires`** and **`ensures`**, respectively. In addition, the ACSL language provides the **`assigns`** clause to specify that a function is not allowed to change

```
/*@
    requires IsValidRange(Bitstream, BitstreamSizeInBytes);
    requires Startposition + Length <=  UINT_MAX;
    requires Length <= 64;
    assigns \nothing;

    behavior out_of_range:
        assumes !ValidBitIndex(Startposition, Length,
           BitstreamSizeInBytes);
        ensures \result == 0;

    behavior normal:
        assumes ValidBitIndex(Startposition, Length,
           BitstreamSizeInBytes);
        ensures \result == BitSum(Startposition, Length, Bitstream);
        ensures !TooBig(\result, Length);

    complete behaviors;
    disjoint behaviors;
*/
uint64_t Bitwalker_Peek(unsigned int Startposition,
                        unsigned int Length,
                        uint8_t Bitstream[],
                        unsigned int BitstreamSizeInBytes);
```

Listing 2.2. Formal specification of `Bitwalker_Peek` in ACSL

memory locations other than the ones explicitly listed. When no **`assigns`** clauses are specified, the function is allowed to modify every visible variable.

The three preconditions for the function arguments of the informal specification are formalized in the function contract by three preconditions. For the first one we use the predicate `IsValidRange` which we specified in ACSL in order to state that the `Bitstream` is a valid array of length `BitstreamSizeInBytes`. Furthermore, we claim that `Bitwalker_Peek` does not alter any memory locations apart from internal function variables via the **`assigns`** clause.

Moreover, we use so-called behaviors in ACSL to describe the two cases from the informal specification. The cases are discriminated through the predicate `ValidBitIndex`. The first behavior `out_of_range` represents the robustness case where the bit sequence is not valid and the second behavior specifies the expected behavior in the normal case.

In both cases we state what the result of peek shall be as postconditions. In addition, we us a negated form of a predicate called `TooBig` in the last postcondition of the normal case. This postcondition was introduced to verify that the functions `Bitwalker_Peek` and `Bitwalker_Poke` interact correctly. Therefore, we will discuss this postcondition in section 2.4.

Since the implementation of `Bitwalker_Peek` contains a loop, we also need a loop specification containing a variant for the termination proof and some invariants to enable the automatic theorem provers to verify the postconditions. Although this loop specification is important for the verification, it is not in the sense to formalize the informal specification.

Since we verify the implementation in accordance to the formal specification, it is crucial that it matches the informal one. Therefore, we reviewed both specifications.

2.2.4 Formal Verification with Frama-C/WP

2.3 The Function `Bitwalker_Poke`

To be done

2.3.1 Informal Specification

The function `Bitwalker_Poke` converts an integer to a bit sequence and writes it into a bit stream. Its function signature reads as follows:

```
int       Bitwalker_Poke(unsigned int Startposition,
                         unsigned int Length,
                         uint8_t Bitstream[],
                         unsigned int BitstreamSizeInBytes,
                         uint64_t Value);
```

The arguments have the following purpose:

- `Startposition` is the bit index in the bit stream where the bit sequence starts.
- `Length` is the length of the bit sequence.
- `Bitstream` is the array which provides the bit stream.
- `BitstreamSizeInBytes` is the length of the array containing the bit stream.
- `Value` is the integer which shall be converted into a bit sequence.

The following preconditions shall hold for the function arguments:

- `Bitstream` is a valid array of length `BitstreamSizeInBytes`
- `Length` < `unsigned int`.
- `Startposition + Length` ≤ `UINT_MAX`.

Note that additional constraints are implicitly expressed by the use of *unsigned* integer types.

Now we can specify `Bitwalker_Poke` as follows: The function `Bitwalker_Poke` converts a 64-bit unsigned integer to a bit sequence and writes it into a bit stream.

For $0 \leq x$ exists a shortest sequence of 0 and 1 $(b_0, b_1, \ldots, b_{n-1})$ such that

$$\sum_{i=0}^{n-1} b_i \cdot 2^{(n-1)-i} = x. \tag{2}$$

The function `Bitwalker_Poke` tries to store the sequence $(b_0, b_1, \ldots, b_{n-1})$ in the bit sequence of `Length` bits that starts at bit index `Startposition`.

The return value of `Bitwalker_Poke` depends on the following three cases:

- If the bit sequence is valid, then there are two cases:
  - If `Length` $\geq n$, then the sequence $(\overbrace{0,\ldots,0}^{\texttt{Length}-n}, b_0, b_1, \ldots, b_{n-1})$ is stored in the bit stream starting at `Startposition`. The return value of `Bitwalker_Poke` is 0.
  - If `Length` $< n$, then the sequence $(b_0, b_1, \ldots, b_{n-1})$ cannot be stored and `Bitwalker_Poke` returns -2.
- If the bit sequence is not valid, then `Bitwalker_Poke` returns -1.

2.3.2 Implementation

Listing 2.3 shows the implementation of `Bitwalker_Poke`. The algorithm consists of three cases. The first two matching the robustness cases of the informal specification (see subsection Informal Specification 2.3.1 on page 9) and the last one writhes the bit stream.

```
int Bitwalker_Poke (unsigned int Startposition, unsigned int Length,
                    uint8_t Bitstream[],
                    unsigned int BitstreamSizeInBytes,
                    uint64_t Value)
{
    if (((Startposition + Length - 1) >> 3) >= BitstreamSizeInBytes)
        return -1;

    uint64_t MaxValue = (((uint64_t)0x01) << Length) - 1;

    if (MaxValue < Value)
        return -2;

    int i;
    for (i = Startposition + Length - 1;
            i >= (int)Startposition; i--)
    {
        if ((Value & 0x01) == 0)
            Bitstream[i >> 3] &= ~BitwalkerBitMaskTable[i & 0x07];
        else
            Bitstream[i >> 3] |=  BitwalkerBitMaskTable[i & 0x07];

        Value >>= 1;
    }
    return 0;
}
```

Listing 2.3. Implementation of `Bitwalker_Poke`

2.3.3 Formal Specification with ACSL

To verify that the implementation meets the informal specification, we need to formalize the it with ACSL. Listing 2.4 shows the translated function contract.

The general preconditions of the informal specification are reflected by the first three **`requires`**-clauses at the beginning of the contract. Because the algorithm modifies the range `Bitstream` and reads the global range `BitwalkerBitMaskTable` we need to express that the two ranges

must use separated memory locations. Therefore we use the predicate `separated` in the fourth **`requires`**-clause. Furthermore, in the following **`assigns`**-clause we must specify the memory locations which altered by the function.

At least, we specify the three behaviors of `Bitwalker_Poke`. The first behavior `out_of_range` occurs if the bit sequence between `Starposition` and `Starposition+Length` not in range `Bitstream`. The postcondition, the return value of the behavior, is formalized with the **`requires`**-clause.

The second behavior `value_too_big` covers the case that `Value` not fits into the given length `Length`.

And finally the behavior `normal` which assumes that the value `Value` is not to big and the bit sequence is in within the range `Bitstream`. Here we assume that the algorithm only writes to the index positions of `Bitstream` between `Starposition` and `Starposition+Length` and all other memory locations which be used by the array are unaltered.

```
/*@
    requires 0 < Length < UINT_MAX;
    requires Startposition + Length <= UINT_MAX;
    requires IsValidRange(Bitstream, BitstreamSizeInBytes);
    requires \separated(Bitstream+(0..BitstreamSizeInBytes-1),
             BitwalkerBitMaskTable+(0..7));

    assigns Bitstream[StreamIndex(Startposition)..
            StreamIndex(Startposition + Length - 1)];

    behavior out_of_range:
        assumes !ValidBitIndex(Startposition,  Length,
                BitstreamSizeInBytes);

        assigns \nothing;

        ensures \result == -1;

    behavior value_too_big:
        assumes TooBig(Value, Length);
        assumes ValidBitIndex(Startposition,  Length,
                BitstreamSizeInBytes);

        assigns \nothing;

        ensures \result == -2;

    behavior normal:
        assumes ValidBitIndex(Startposition, Length,
                BitstreamSizeInBytes);
        assumes !TooBig(Value, Length);

        assigns Bitstream[StreamIndex(Startposition)..
                StreamIndex(Startposition + Length - 1)];

        ensures BitSum(Startposition, Length, Bitstream) == Value;
        ensures BitSum(0, Startposition, \old(Bitstream))
                == BitSum(0, Startposition, Bitstream);
        ensures BitSum(Startposition+Length, BitstreamSizeInBytes,
                \old(Bitstream)) == BitSum(Startposition+Length,
                BitstreamSizeInBytes, Bitstream);
        ensures \result == 0;

    complete behaviors;
    disjoint behaviors;
*/

int      Bitwalker_Poke(unsigned int Startposition,
                        unsigned int Length,
                        uint8_t Bitstream[],
                        unsigned int BitstreamSizeInBytes,
                        uint64_t Value);
```

Listing 2.4. Formal Specification of `Bitwalker_Poke`

2.3.4 Formal Verification with Frama-C/WP

2.4 Interaction of `Bitwalker_Peek` and `Bitwalker_Poke`

To be done

2.4.1 Informal Specification

The functions `Bitwalker_Peek` and `Bitwalker_Poke` are inverse to each other.

2.4.2 Implementation

2.4.3 Formal Specification with ACSL

2.4.4 Formal Verification with Frama-C/WP

2.5 Open Issues

To be done

3 SQS

4 CEA LIST

5 Systerel

6 Conlusions