OETCS/WP4/D4.2.2

openETCS

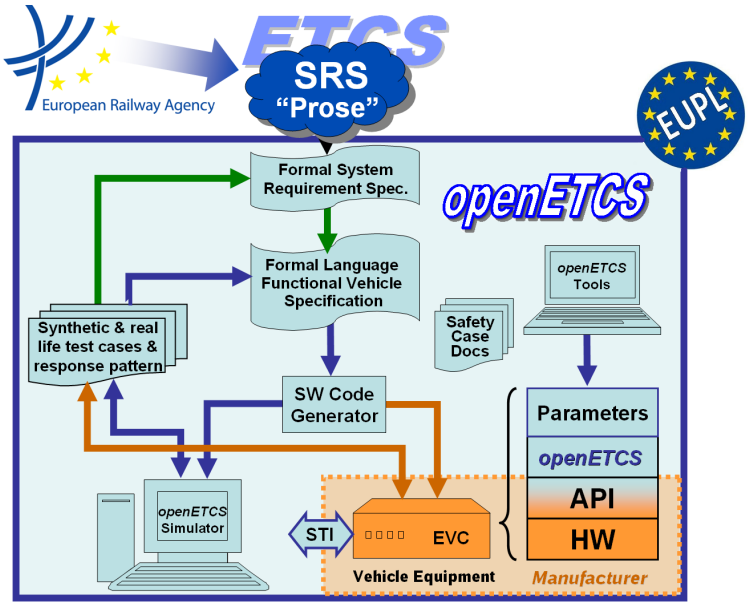
ITEA2 Project
Call 6 11025
2012 – 2015

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

Figures and Tables

Figures

Tables

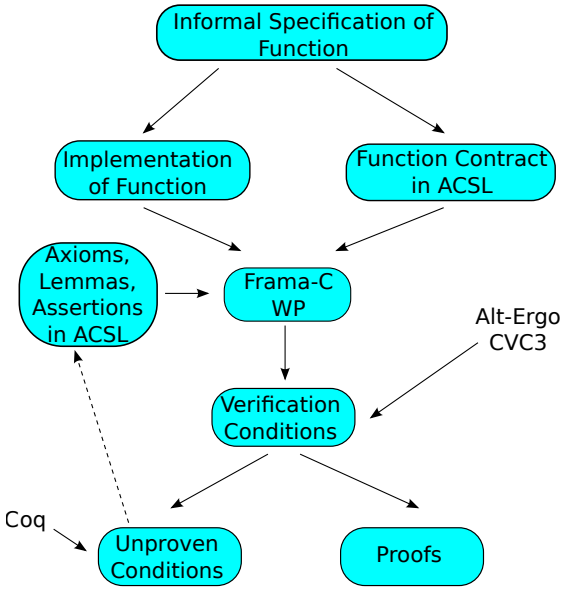


Figure 2.1. The method to formally verify the Bitwalker.

1 Introduction

2 Formal Verification of Bitwalker

In this section we describe our work on the formal verification of the so-called Bitwalker. The Bitwalker shall read bit sequences from a bit stream and convert them to an integer. Furthermore, it shall convert an integer into a bit sequence and write it into a bit stream. Therefore, the Bitwalker has a read and a write function, namely `Bitwalker_Peek` and `Bitwalker_Poke`.

Our aim is to verify the functionality of `Bitwalker_Peek` and `Bitwalker_Poke` as well as their correct interaction. Furthermore, we want to verify some robustness cases for `Bitwalker_Peek` and `Bitwalker_Poke` and the absence of run time errors for both functions. We won't take into account any complexity requirements.

We introduce a method to achieve these goals in section 2.1. Moreover, it is our intention to elaborate the method and in particular the associated tools.

Subsequently, we use the method for `Bitwalker_Peek` and `Bitwalker_Poke` in section 2.2 and 2.3, respectively. We provide an informal specification, an implementation and a formal specification for each function and present what could have been verified for the implementation.

We discuss the interaction of these functions in section 2.4 where we show how the interaction can be formally specified and present the verification results. Finally, we give an overview about the still open issues in section 2.5.

2.1 Verification Method

In this section we introduce our method of choice along with the used tools. We use a deductive verification approach to formally prove that a function fulfills its specification. The foundations for deductive verification are axiomatic semantics as formulated by Hoare [1]. Figure 2.1 shows the method with the involved verification tools.

Starting point is an informal specification of a function with which in mind a implementation is written and on which basis the formal specification is created. The formal specification of a function is a so-called function contract which contains preconditions to express what a function expects from its caller and postconditions to state the guarantees after the execution. The specification language is called ACSL (ANSI/ISO-C Specification Language) [2] which is a formal language to express behavioral properties of C programs.

Moreover, it is the specification language associated with the verification platform Frama-C [3] which we use along with its plug-in WP [4]. Within Frama-C, WP enables the deductive verification of C programs that have been annotated with ACSL. WP generates verification conditions which are submitted to external automatic or interactive theorem provers. A function is then verified if each verification condition is discharged by at least one prover.

Figure 2.1 shows that we first apply the automatic theorem provers Alt-Ergo [5] and CVC3 [6] and then apply the interactive theorem prover Coq [7] for the still unproven conditions in order to automate as much as possible. Moreover, unproven conditions motivate to give some extra information in the form of axioms, lemmas and assertions in ACSL, since they can ease the search of a proof. One need to be careful with axioms because they can yield contradictions and thus make the proof system unsound. This is different for lemmas and assertions because WP will generate additional verification conditions for them.

In order to prove the absence of run time errors we use the `rte` option of WP that automatically introduce ACSL assertions. If all these assertions can be proven, then the absence of run time errors is guaranteed.

2.2 The Function `Bitwalker_Peek`

In this section we examine the function `Bitwalker_Peek`. Initially, we provide an informal specification followed by an implementation. We then derive a formal specification on the basis of the informal one. Finally, we present the results of the deductive verification with Frama-C and WP.

2.2.1 Informal Specification

We first 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 2.2 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.

- A *bit sequence* is a consecutive sequence of bits within a bit stream as represented in Figure 2.3.

  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.

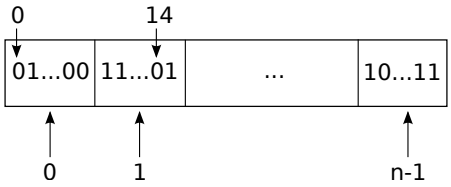


Figure 2.2. Array indices and bit indices in a bit stream

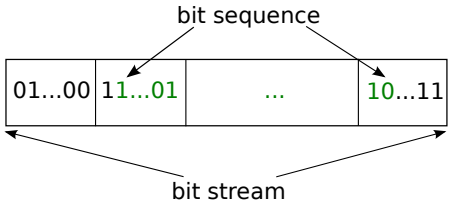


Figure 2.3. A bit sequence within a bit stream

- 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`.

```
/*@
    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

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 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 straight forward in the function contract also 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 for a distinction of the two cases from the informal specification. The cases are discriminated through the predicate `ValidBitIndex` which indicates whether a bit sequence is valid or not. 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 `Bitwalker_Peek` shall be as postconditions. In addition, we use 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 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 respect to the formal specification, it is crucial that it matches the informal one. Therefore, we reviewed the accordance of both specifications.

2.2.4 Formal Verification with Frama-C/WP

In this section we present the current state of the verification results for `Bitwalker_Peek`. Table 2.1 discriminates the results for three different types of verification conditions (VCs).

The first row contains the lemmas we used to ease the verification for the automatic theorem provers. While the second row contains the `rte`-assertions concerning the absence of run time errors. The third row shows all other verification conditions for `Bitwalker_Peek` which are mainly about for correct functional behavior. However, they also contain the postconditions for the robustness cases and the loop specification.

For each row we listed the total number of generated verification conditions, the number of proven verification conditions and the verification rate that is the percentage of proven verification conditions.

The verification rate for the `rte`-assertions are very low due to the difficulty for Frama-C to deal with bit operations. In order to increase this rate, we will verify the absence of run time errors separately and will provide additional lemmas and axioms to ease the verification. We point out some of the related challenges in section 2.5.

	# VC	Proven VCs	Verification rate in %
lemmas	1	0	0
rte-assertions	9	5	55
rest	18	17	94

Table 2.1. Verification Results of `Bitwalker_Peek`

2.3 The Function `Bitwalker_Poke`

In this section we examine the function `Bitwalker_Poke` in the same manner as we did it for `Bitwalker_Peek` in section 2.2.

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` which discriminates three cases. The first two are the robustness cases of the informal specification and the last one is the normal case where the function actually writes into the bit stream. Similar to `Bitwalker_Peek` a lot of bit operations are used.

```
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

Listing 2.4 shows the function contract of `Bitwalker_Poke`. The case independent preconditions of the informal specification are reflected by the first three **`requires`**-clauses at the beginning of the contract. `Bitwalker_Poke` modifies the `Bitstream` and reads the array `BitwalkerBitMaskTable` thus we need to express that the two arrays must have separated memory locations. Therefore, we use the predicate `separated` in the fourth **`requires`**-clause.

Furthermore, in the following **`assigns`**-clause we specify the memory locations which can be altered by the function.

We specify the three cases of `Bitwalker_Poke` by using behaviors. The first behavior `out_of_range` occurs if the given bit sequence is not valid with respect to the `Bitstream`. The second behavior `value_too_big` covers the case that the value `Value` is not representable with only `Length` bits.

Finally, the behavior `normal` assumes that `Value` is not too big and the bit sequence is valid. Here, `Bitwalker_Poke` writes the particular bit sequence into `Bitstream` while all other memory locations are unaltered. For all behaviors there is one postcondition to state what the return value shall be in this case.

```
/*@
    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

In this section we present the current state of verification results of for `Bitwalker_Poke`. The results are shown in Table 2.2. We listed the different verification conditions row by row like we did it for `Bitwalker_Peek`.

The function `Bitwalker_Poke` has significantly more unproven verification conditions than `Bitwalker_Peek` this is because it is more complex and alters memory locations via bit operations. Therefore, we will verify the absence of run time errors separately as well.

	# VC	Proven VCs	Proven VCs in %
lemmas	1	0	0
rte-assertions	19	7	36
rest	49	38	77

Table 2.2. Verification Results of `Bitwalker_Poke`

2.4 Interaction of `Bitwalker_Peek` and `Bitwalker_Poke`

In this section we examine the interaction of `Bitwalker_Peek` and `Bitwalker_Poke`. The functions shall be inverse to each other in respect to the normal cases of both functions. In the following we provide a formal specification and present our verification results.

2.4.1 Formal Specification with ACSL and C

For the specification of the interaction we have two auxiliary C functions: The first one to call first `Bitwalker_Poke` on a bit stream and then `Bitwalker_Peek` and the second one to do it the other way around.

Figure 2.5 shows the straight forward implementation of the first helper function along with its ACSL contract. The contract contains a lot of preconditions because we only specify an interaction for the case that both functions are in their normal cases. The reader can compare the preconditions with them from the contracts of `Bitwalker_Peek` and `Bitwalker_Poke` with respect to the `normal` behaviors. As a postcondition we formulate that `Bitwalker_Peek` reads exactly the value written by `Bitwalker_Poke`.

```
/*@
    requires 0 < Length < UINT_MAX;
    requires Startposition + Length <=  UINT_MAX;
    requires IsValidRange(Bitstream, BitstreamSizeInBytes);
    requires !TooBig(Value, Length);
    requires \separated(Bitstream+(0..BitstreamSizeInBytes-1), &
       BitwalkerBitMaskTable[0..7]);
    requires ValidBitIndex(Startposition, Length,
       BitstreamSizeInBytes);
    requires Length <= 64;

    ensures \result == Value;
*/
uint64_t peek_poke_inverse (unsigned int Startposition, unsigned int
    Length, uint8_t Bitstream[],
    unsigned int BitstreamSizeInBytes, uint64_t Value)
{
    Bitwalker_Poke(Startposition, Length, Bitstream,
       BitstreamSizeInBytes, Value);
    //@ assert BitSum(Startposition, Length, Bitstream) == Value;
    return Bitwalker_Peek(Startposition, Length, Bitstream,
       BitstreamSizeInBytes);
}
```

Listing 2.5. Specification of interaction when first calling `Bitwalker_Poke`.

Figure 2.6 shows the straight forward implementation of the second auxiliary function along with its ACSL contract. In contrast to the first function, we work with two bit streams. With `Bitwalker_Peek` a certain bit sequence is read out from one bit stream and written via `Bitwalker_Poke` into the other one. Therefore, we formulate as a postconditions that the two bit streams are equal in these certain ranges.

```
/*@
    requires 0 < Length < UINT_MAX;
    requires Startposition1 + Length <=  UINT_MAX;
    requires Startposition2 + Length <=  UINT_MAX;
    requires IsValidRange(Bitstream1, BitstreamSizeInBytes1);
    requires IsValidRange(Bitstream2, BitstreamSizeInBytes2);
    requires \separated(Bitstream2+(0..BitstreamSizeInBytes2-1), &
       BitwalkerBitMaskTable[0..7]);
    requires \separated(Bitstream1+(0..BitstreamSizeInBytes1-1),
       Bitstream2 + (0..BitstreamSizeInBytes2-1));
    requires Length <= 64;
    requires ValidBitIndex(Startposition1, Length,
       BitstreamSizeInBytes1);
    requires ValidBitIndex(Startposition2, Length,
       BitstreamSizeInBytes2);

    ensures BitSum{Old}(Startposition1, Length, Bitstream1) ==
       BitSum(Startposition2, Length, Bitstream2);
*/
void poke_peek_inverse (unsigned int Startposition1, unsigned int
   Startposition2,
    unsigned int Length, uint8_t Bitstream1[], uint8_t Bitstream2[],
    unsigned int BitstreamSizeInBytes1, unsigned int
       BitstreamSizeInBytes2)
{
    uint64_t x = Bitwalker_Peek(Startposition1, Length, Bitstream1,
       BitstreamSizeInBytes1);

    Bitwalker_Poke(Startposition2, Length, Bitstream2,
       BitstreamSizeInBytes2, x);
}
```

Listing 2.6. Specification of interaction when first calling `Bitwalker_Peek`.

2.4.2 Formal Verification with Frama-C/WP

All postconditions (and the assertion in figure 2.6) are verified. Therefore, we succeeded to prove that the functions `Bitwalker_Peek` and `Bitwalker_Poke` interact correctly in respect to their specifications.

Moreover, this verification result serves as a validation for the contracts of `Bitwalker_Peek` and `Bitwalker_Poke` because the verification of the interaction depends on these contracts. Since both functions are called, the caller has to assure that all preconditions hold and can then rely on the guarantees given by the postconditions.

As a remark, we extended the contract of `Bitwalker_Peek` by one postcondition to ease the verification of the interaction. The postcondition simply states that the read value is always representable by `Length` bits which is obviously always the case, since the value is the sum of `Length` bits. The postcondition is needed to assure that the preconditions for the normal case of `Bitwalker_Poke` are fulfilled when first calling `Bitwalker_Peek` and then `Bitwalker_Poke`.

2.5 Open Issues

Previously, we have seen that WP cannot yet deal very well with bit operations due to the fact that WP's memory models do not provide much information about bit operations. Therefore, the provers have less options to manipulate the proof goal. This problem is known and CEA LIST is working on a solution for the next release.

As a work around one could introduce axioms which provide additional informations about bit operations. The problem by using axioms is that one can introduce wrong facts yielding contradictions which make the whole proof system unsound. Thus, it is an approach one should be really careful with.

Moreover, the chosen automatic theorem provers are generally not very good when it comes to mixing arithmetic and bit operations. However, there is an automatic theorem prover namely Z3 which is stronger with this. At the moment Frama-C does not provide an interface for Z3 but this will probably change with future releases. Thus, we can expect a better automatic verification rate here.

In addition, we can use the interactive theorem prover Coq to verify unproven verification conditions. Herewith, we can search a proof for a verification condition on our own by using Coq's various support mechanisms. Nevertheless, Coq has only the informations provided by the memory model and the user like the other provers.

3 SQS

4 CEA LIST

5 Systerel

6 Conlusions

References

[1] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, 1969.

[2] ANSI/ISO C Specification Language. `http://frama-c.com/acsl.html`.

[3] Frama-C Software Analyzers. `http://frama-c.com`.

[4] WP Plug-in. `http://frama-c.com/wp.html`.

[5] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Homepage of the Alt-Ergo Theorem Prover. `http://alt-ergo.lri.fr/`.

[6] Clark Barrett and Cesare Tinelli. Homepage of CVC3. `http://www.cs.nyu.edu/acsys/cvc3/`, 2010.

[7] Coq Development Team. *The Coq Proof Assistant Reference Manual*, v8.3 edition, 2011. `http://coq.inria.fr/`.

[8] Microsoft Research. Homepage of the Z3 SMT Solver. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.