OETCS/WP4/D4.2.2

openETCS

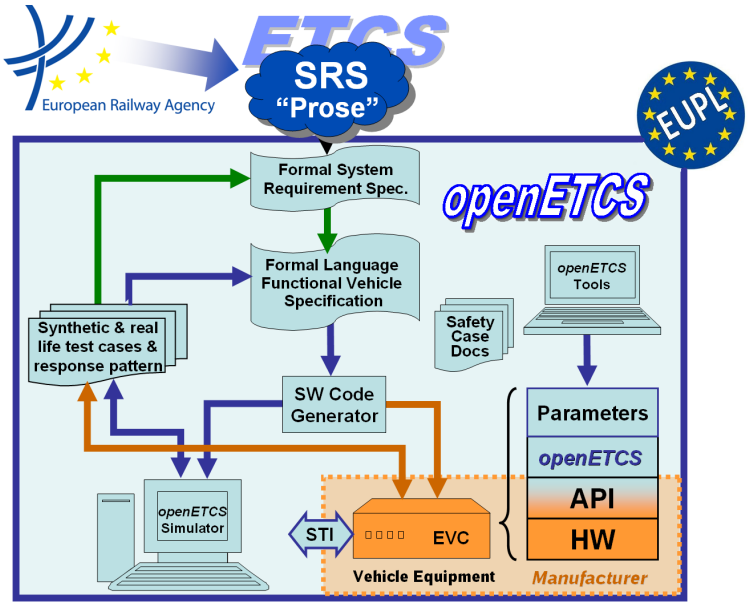
ITEA2 Project
Call 6 11025
2012 – 2015

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 6
2.1 Verification Objectives 6
2.2 The Function `Bitwalker_Peek` 6
2.3 The Function `Bitwalker_Poke` 9
2.4 Interaction of `Bitwalker_Peek` and `Bitwalker_Poke` 14
2.5 Open Issues 14
3 SQS 15
3.1 Introduction 15
3.2 Resource Standard Metrics -RSM- Results 15
3.3 LocMetrics tool Results 19
3.4 Understand tool Results 20
3.5 Clang Static Analyzer tool Results 23
3.6 CPPcheck tool Results 25
3.7 Results Comparison 26
4 Conlusions 27

Figures and Tables

Figures

Figure 1. Array indices and bit indices in a bit stream 6
Figure 2. A bit sequence within a bit stream 7
Figure 3. MISRA-C Rules results 23
Figure 4. Clang Analysis results 25
Figure 5. cppcheck results 26

Tables

Table 1. Quality Notices 15
Table 1. Quality Notices 16
Table 1. Quality Notices 17
Table 1. Quality Notices 18
Table 2. Quality Profile 19
Table 3. LocMetrics Tool Results 20
Table 4. Ubication of MISRA Violations 23
Table 5. xxx 24
Table 5. xxx 25

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.

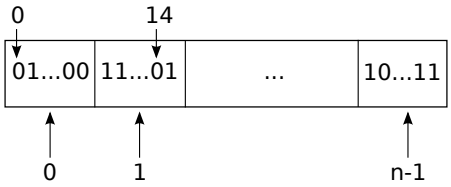


Figure 1. Array indices and bit indices in a bit stream

- 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.

- 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

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

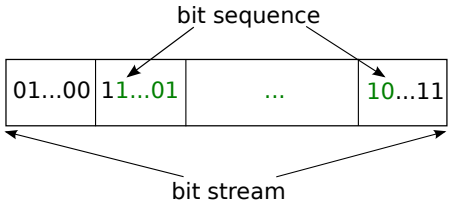


Figure 2. A bit sequence within a bit stream

- 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 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.

```
/*@
    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

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

3.1 Introduction

include static analysis info and the tasks done by SQS

3.2 Resource Standard Metrics -RSM- Results

Resource Standard Metrics (RSM) is a source code metrics and quality analysis tool.

RSM provides standard metrics and a combination of features that allow to:

- Analyze source code for programming errors
- Analyze source code for code style enforcement
- Create an Inheritance tree from the code
- Collect Source Code Metrics by the function, class, file, and project
- Analyze Cyclomatic Complexity

RSM tool is mapped to the [MISRA C] Industry Standards which coverage is 40.16%

Besides, RSM has intrinsic quality notices and can be extended by the end user with User Defined Quality Notices using regular expressions to analyze code lines.

The following table shows the intrinsic Quality Notices for language c.

Table 1. Quality Notices

Quality Notice No. 1 Emit a quality notice when the physical line length is greater than the specified number of characters. Rationale: Reproducing source code on devices that are limited to 80 columns of text can cause the truncation of the line or wrap the line. Wrapped source lines are difficult to read, thus creating weaker peer reviews of the source code.	**Quality Notice No. 2** Emit a quality notice when the function name length is greater than the specified number of characters. Rationale: Long function names may be a portability issue especially when code has to be cross compiled onto embedded platforms. This difficultly is typically seen with older hardware and operating systems.
Quality Notice No. 3 Emit a quality notice when ellipsis '...' are identified within a functions parameter list thus enabling variable arguments. Rationale: Ellipsis create a variable argument list. This type of design is found in C and C++. It essentially breaks the type strict nature of C++ and should be avoided.	**Quality Notice No. 4** Emit a quality notice if there exists an assignment operator '=' within a logical 'if' condition. Rationale: An assignment within an "if" condition is likely a typographical error giving rise to a logic defect. However, some programmers place compound statements into the "if" condition making the code difficult to read.

Table 1. Quality Notices

Quality Notice No. 5 Emit a quality notice if there exists an assignment operator '=' within a logical 'while' condition. Rationale: An assignment within a "while" condition is likely a typographical error giving rise to a logic defect. However, some programmers place compound statements into the "while" condition making the code difficult to read.	**Quality Notice No. 6** Emit a quality notice when a pre-decrement operator '–' is identified within the code. Rationale: The pre-decrement of a variable occurs before the remainder of the processing in the statement. This can be difficult to comprehend or anticipate. There are documented cases where the mathematical results vary between the result of macros when different code preprocessors expand the macros into a normal form. Remember, there is no standard for the preprocessor, just the language.
Quality Notice No. 7 Emit a quality notice when a pre-increment operator '++' is identified within the code. Rationale: The pre-increment of a variable occurs before the remainder of the processing in the statement. This can be difficult to comprehend or anticipate. There are documented cases where the mathematical results vary between the result of macros when different code preprocessors expand the macros into a normal form.	**Quality Notice No. 8** Emit a quality notice when the 'realloc' function is identified within the code. Rationale: Using realloc can lead to latent memory leaks within your C or C++ code. The call to realloc reassigns the pointer to the same memory address using a larger or smaller space. However if realloc fails, a NULL pointer is returned. No "free" was performed on the pointer so if you don't retain the pointer before the realloc call, a latent memory leak could occur.
Quality Notice No. 9 Emit a quality notice when the 'goto' function is identified within the code. Rationale: The use of "goto" creates spaghetti code. A "goto" can jump anywhere to the destination label. This type of design breaks the "one in - one out" ideal of a function creating code which can be impossible to debug or maintain.	**Quality Notice No. 10** Emit a quality notice when the Non-ANSI function prototype is identified within the code. Rationale: Older C code can be written in a style that does not use function prototypes of the function argument types. This code will not compile on ANSI C and C++ compilers because of this type of weakness. Identifying this condition can help assess whether code can be ported to a newer version of the language.
Quality Notice No. 11 Emit a quality notice when open and closed brackets '[ ]' are not balance within a file. Rationale: This type of error is always caught by the compiler as a syntax error. However, a compiler can be told to ignore source code by using preprocessor directives like #if ... #endif. This is a way to "comment" out large blocks of code. However, the code still looks like operational code to the maintainer as it is not a comment. Many hours can be wasted working on dead code. This quality notice serves to warn you of this dead code that should be removed or converted to actual comment form.	**Quality Notice No. 12** Emit a quality notice when open and closed parenthesis '( )' are not balance within a file. Rationale: This type of error is always caught by the compiler as a syntax error. However, a compiler can be told to ignore source code by using preprocessor directives like #if ... #endif. This is a way to "comment" out large blocks of code. However, the code still looks like operational code to the maintainer as it is not a comment. Many hours can be wasted working on dead code. This quality notice serves to warn you of this dead code that should be removed or converted to actual comment form..
Quality Notice No. 13 Emit a quality notice when a 'switch' statement does not have a 'default' condition. Rationale: A "switch" statement must always have a default condition or this logic construct is nondeterministic. Generally the default condition should warn the user of an anomalous condition which was not anticipated by the programmer by the case clauses of the switch.	**Quality Notice No. 14** Emit a quality notice when there are more 'case' conditions than 'break', 'return' or 'fall through' comments. Rationale: Many tools, including RSM, watch the use of "case" and "break" to insure that there is not an inadvertent fall through to the next case statement. RSM requires the programmer to explicitly indicate in the source code via a "fall through" comment that the case was designed to fall through to the next statement.

Table 1. Quality Notices

Quality Notice No. 16 Emit a quality notice when function white space percentage is less than the specified minimum. Rational: Source code must be easily read. A low percentage of white space indicates that the source code is crammed together thus compromising the readability of the code. Typically white space less than 10 percent is considered crammed code.	**Quality Notice No. 17** Emit a quality notice when function comment percentage is less than the specified minimum. Rationale: A programmer must supply sufficient comments to enable the understandability of the source code. Typically a comment percentage less than 10 percent is considered insufficient. However, the content quality of the comment is just as important as the quantity of the comments. For this reason you could use the -E option to extract all the comments from a file. The reviewer should be able to read the comments and extract the story of the code.
Quality Notice No. 18 Emit a quality notice when the eLOC within a function exceeds the specified maximum. Rationale: An extremely large function is very difficult to maintain and understand. When a function exceeds 200 eLOC (effective lines of code), it typically indicates that the function could be broken down into several functions. Small modules are desirable for modular composability.	**Quality Notice No. 19** Emit a quality notice when file white space percentage is less than the specified minimum. Rationale: Source code must be easily read. A low percentage of white space indicates that the source code is crammed together thus compromising the readability of the code. Typically white space less than 10 percent is considered crammed code.
Quality Notice No. 20 Emit a quality notice when file comment percentage is less than the specified minimum. Rationale: A programmer must supply sufficient comments to enable the understandability of the source code. Typically a comment percentage less than 10 percent is considered insufficient. However, the content quality of the comment is just as important as the quantity of the comments. For this reason you could use the -E option to extract all the comments from a file. The reviewer should be able to read the comments and extract the story of the code.	**Quality Notice No. 22** Emit a quality notice when each if, else, for or while is not bound by scope. Rationale: Logical blocks should be bound with scope. This clearly marks the boundaries of scope for the logical blocks. Many times, code may be added to non-scoped logic blocks thus pushing other lines of code from the active region of the logical construct giving rise to a logic defect.
Quality Notice No. 23 Emit a quality notice when the '?' or the implied if-then-else construct has been identified. Rationale: The ? operator creates the code equivalent of an "if" then "else" construct. However the resultant source is far less readable.	**Quality Notice No. 24** Emit a quality notice when an ANSI C++ keyword is identified within a *.c or a *.h file. Rationale: In C source code it is possible to find variable names like "class". This word is a key word in C++ and would prevent this C code from being ported to the C++ language.
Quality Notice No. 25 (Deprecated RSM 6.70) When analyzing *.h files for C++ keywords, assume that *.h can be both C and C++. Rationale: A *.h file can be either a C or C++ source file. If a *.h file is assumed to be from either language, then RSM will not emit C keyword notices in *.h file, only for *.c files.	**Quality Notice No. 26** Emit a quality notice when a void * is identified within a source file. Rationale: A "void *" is a type-less pointer. ANSI C and C++ strives to be type strict. In C++ a "void *" breaks the type strict nature of the language which can give rise to anomalous run-time defects.
Quality Notice No. 27 Emit a quality notice when the number of function return points is greater than the specified maximum. Rationale: A well constructed function has one entry point and one exit point. Functions with multiple return points are difficult to debug and maintain.	**Quality Notice No. 28** Emit a quality notice when the cyclomatic complexity of a function exceeds the specified maximum. Rationale: Cyclomatic complexity is an indicator for the number of logical branches within a function. A high degree of V(g), greater than 10 or 20, indicates that the function could be broken down into a more modular design of smaller functions.

Table 1. Quality Notices

Quality Notice No. 29 Emit a quality notice when the number of function input parameters exceeds the specified maximum. Rationale: A high number of input parameters to a function indicates poor modular design. Data should be grouped into representative data types. Functions should be specific to one purpose.	**Quality Notice No. 30** Emit a quality notice when a TAB character is identified within the source code. Indentation with TAB will create editor and device dependent formatting. Rationale: Tab characters within source code create documents that are print and display device dependent. The document may look correct on the screen but it may become unreadable when printed.
Quality Notice No. 31 Emit a quality notice when class comment percentage is less than the specified minimum. Rationale: A programmer must supply sufficient comments to enable the understandability of the source code. Typically a comment percentage less than 10 percent is considered insufficient.	**Quality Notice No. 43** Emit a quality notice when the key word 'continue' has been identified within the source code. Rationale: The use of 'continue' in logical structures causes a disruption in the linear flow of the logic. This style of programming can make maintenance and readability difficult.
Quality Notice No. 46 Emit a quality notice when function, struct, class or interface blank line percentages are less than the specified minimum Rationale: The amount of blank lines in a file can indicate the degree of readability in the file. It indicates the author intended his work to be human consumable.	**Quality Notice No. 47** Emit a quality notice when the file blank line percentage is less than the specified minimum Rationale: The amount of blank lines in a file can indicate the degree of readability in the file. It indicates the author indented his work to be human consumable.
Quality Notice No. 48 Emit a quality notice when a function has no logical lines of code. Rationale: This condition indicates a no-op or stubbed out function with no operational code.Many code generators create such no-op functions which contribute to code bloat and unnecessary resource utilization.	**Quality Notice No. 49** Emit a quality notice when a function has no parameters in the parameter list. Rationale: A function should always specify the actual parameter names to enhance maintenance and readability. A programmer should always put void to indicate the deliberate design in the code.
Quality Notice No. 50 Emit a quality notice when a variable is assigned to a literal value. Configurable for literal 0 in rsm.cfg. Rationale: A symbolic constant is the preferred method for variable assignment as this creates maintainable and understandable.	**Quality Notice No. 51** Emit a quality notice when there is no comment before a function block. Rationale: A function block should retain a preceding comment block describing the purpose, parameters, returns and algorithms.
Quality Notice No. 52 Emit a quality notice when there is no comment before a class block. Rationale: A class block should retain a preceding comment block describing the purpose, and algorithms.	**Quality Notice No. 53** Emit a quality notice when there is no comment before a struct block. Rationale: A struct block should retain a preceding comment block describing the data and purpose.
Quality Notice No. 55 Emit a quality notice when scope exceeds the specified maximum in the rsm.cfg file. Rationale: A deep scope block of complex logic or levels may indicate a maintenance concern.	**Quality Notice No. 56** Emit a quality notice when sequential break statements are identified. Rationale: Repetitive and sequential breaks can be used to fool RSM identification of case statement without breaks.

In addition to this, RSM allows to customize the desired output providing standard metrics and a combination of features.

RSM has been customized to obtain the below metrics and analysis and the corresponding reports that are available into the [VnVUserStories folder]

- Project Functional Metrics and Analysis
- Project Class/Struct Metrics and Analysis
- Class Inheritance Tree
- Project Quality Profile
- Quality Notice Density
- Files Keywords and Metrics
- Project Keywords and Metrics
- Files Function Metrics
- Class/Struct Metrics
- Complexity Metrics

To be included summary results .....

Table 2. Quality Profile

Type	Count	Percent	Quality Notice
1	38	9.57	Physical line length > 80 characters
2	4	1.01	Function name length > 32 characters
22	5	1.26	if, else, for or while not bound by scope
27	2	0.50	Number of function return points > 1
30	330	83.12	TAB character has been identified
50	7	1.76	Variable assignment to a literal number
51	8	2.02	No comment preceding a function block
53	1	0.25	No comment preceding a struct block
125	2	0.50	A data member in the header file is not of the form m_*

3.3 LocMetrics tool Results

LocMetrics counts total lines of code (LOC), blank lines of code (BLOC), comment lines of code (CLOC), lines with both code and comments (C&SLOC), logical source lines of code (SLOC-L), McCabe VG complexity (MVG), Header Comments (HCLOC), Header Words (HCWORD) and number of comment words (CWORDS). Physical executable source lines of code (SLOC-P) is calculated as the total lines of source code minus blank lines and comment lines. Counts are calculated on a per file basis and accumulated for the entire project. LocMetrics also generates a comment word histogram.

About the results obtained by LocMetrics tool are the following ones:

Table 3. LocMetrics Tool Results

File	LOC	SLOC-P	SLOC-L	MVG	BLOC	C&SLOC	CLOC	CWORD	HCLOC	HCWORD
Bitwalker.h	42	15	12	0	8	1	19	102	0	0
Bitwalker.c	110	58	36	15	24	5	28	217	0	0
main.c	128	45	26	1	23	5	60	350	0	0
opnETCS.h	1250	884	883	0	181	637	185	3864	0	0
opnETCS _Decoder.h	85	62	61	0	3	0	20	103	0	0

3.4 Understand tool Results

Understand is a cross-platform, multi-language, maintenance-oriented IDE (Interactive Development Environment). It is designed to help maintain and understand large amounts of legacy or newly created source code. With this tool SQS has checked MISRA-C:2004

Below the MISRA-C tested rules are listed:

- **Language extensions**
  - 2.1 (req): Assembly language shall be encapsulated and isolated.
  - 2.2 (req): Source code shall only use /* ... */ style comments.
  - 2.3 (req): The character sequence /* shall not be used within a comment.
  - 2.4 (adv-): Sections of code should not be 'commented out'.
- **Character sets**
  - 4.1 (req): Only those escape sequences that are defined in the ISO C standard shall be used.
  - 4.2 (req): Trigraphs shall not be used. Identifiers
- **Identifiers**
  - 5.1 (req): Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
  - 5.2 (req): %s: Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
  - 5.3 (req-): A 'typedef' name shall be a unique identifier.
  - 5.4 (req): A tag name shall be a unique identifier.
  - 5.5 (adv-): No object or function identifier with static storage duration should be reused.
  - 5.6 (adv-): No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.
  - 5.7 (adv-): No identifier name should be reused.
- **Types**

  - 6.3 (adv): 'typedefs' that indicate size and signedness should be used in place of the basic types.
  - 6.4 (req): Bit fields shall only be defined to be of type 'unsigned int' or 'signed int'.
  - 6.5 (req-): Bit fields of type signed int shall be at least 2 bits long.

- **Constants**
  - 7.1 (req): Octal constants (other than zero) and octal escape sequences shall not be used.

- **Declarations and definitions**
  - 8.5 (req-): There shall be no definitions of objects or functions in a header file.
  - 8.6 (adv): Functions shall be declared at file scope.
  - 8.7 (req): Objects %s shall be defined at block scope if they are only accessed from within a single function %s.
  - 8.8 (req): %s: An external object or function shall be declared in one and only one file.
  - 8.9 (req): %s: An identifier with external linkage shall have exactly one external definition.
  - 8.10 (req): %s: All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
  - 8.11 (req): The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.

- **Initialisation**
  - 9.3 (req): In an enumerator list, the '=' construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised.

- **Control statement expressions**
  - 13.3 (req): Floating-point expressions shall not be tested for equality or inequality.

- **Control flow**
  - 14.1 (req-): There shall be no unreachable code.
  - 14.3 (req-): Before preprocessing, a null statement shall only occur on a line by itself;, it may be followed by a comment provided that the first character following the null statement is a white-space character.
  - 14.4 (req): The 'goto' statement shall not be used.
  - 14.5 (req): The 'continue' statement shall not be used.
  - 14.7 (req): A function shall have a single point of exit at the end of the function.
  - 14.10 (req): All 'if ... else if' constructs shall be terminated with an 'else' clause.

- **Switch statements**
  - 15.3 (req): The final clause of a 'switch' statement shall be the 'default' clause.

- **Functions**
  - 16.1 (req): Functions shall not be defined with variable numbers of arguments.
  - 16.2 (req): Functions shall not call themselves, either directly or indirectly.

- 16.3 (req): Identifiers shall be given for all of the parameters in a function prototype declaration.
- 16.4 (req-): The identifiers used in the declaration and definition of a function shall be identical.
- 16.5 (req): Functions with no parameters shall be declared with parameter type void.

- **Pointers and arrays**
  - 17.5 (adv): The declaration of objects should contain no more than 2 levels of pointer indirection.

- **Structures and unions**
  - 18.4 (req): Unions shall not be used.

- **Preprocessing directives**
  - 19.1 (adv-): '#include' statements in a file should only be preceded by other preprocessor directives or comments.
  - 19.2 (adv): Non-standard characters should not occur in header file names in include directives.
  - 19.3 (req): The '#include' directive shall be followed by either a <filename> or f̈ilenameës̈equence.
  - 19.4 (req-): C macros shall only expand to a braced initialised, a constant, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
  - 19.5 (req): Macros shall not be '#define'd or '#undef'd within a block.
  - 19.6 (req): '#undef' shall not be used.

- **Standard libraries**
  - 20.4 (req): Dynamic heap memory allocation shall not be used.
  - 20.5 (req): The error indicator 'errno' shall not be used.
  - 20.6 (req): The macro 'offsetof', in library <stddef.h>, shall not be used.
  - 20.7 (req): The 'setjmp' macro and the 'longjmp' function shall not be used.
  - 20.8 (req): The signal handling facilities of <signal.h> shall not be used.
  - 20.9 (req): The input/output library <stdio.h> shall not be used in production code.
  - 20.10 (req): The library functions 'atof', 'atoi' and 'atol' from library <stdlib.h> shall not be used.
  - 20.11 (req): The library functions 'abort', 'exit', 'getenv' and 'system' from library <stdlib.h> shall not be used.
  - 20.12 (req): The time handling functions of library <time.h> shall not be used.

- **Run-time failures**
  - 21.1 (req-): Minimization of run-time failures shall be ensured by the use of at least one of:
    * static analysis tools/techniques;
    * dynamic analysis tools/techniques;
    * explicit coding of checks to handle run-time faults.

The results of the MISRA Rules are the following:

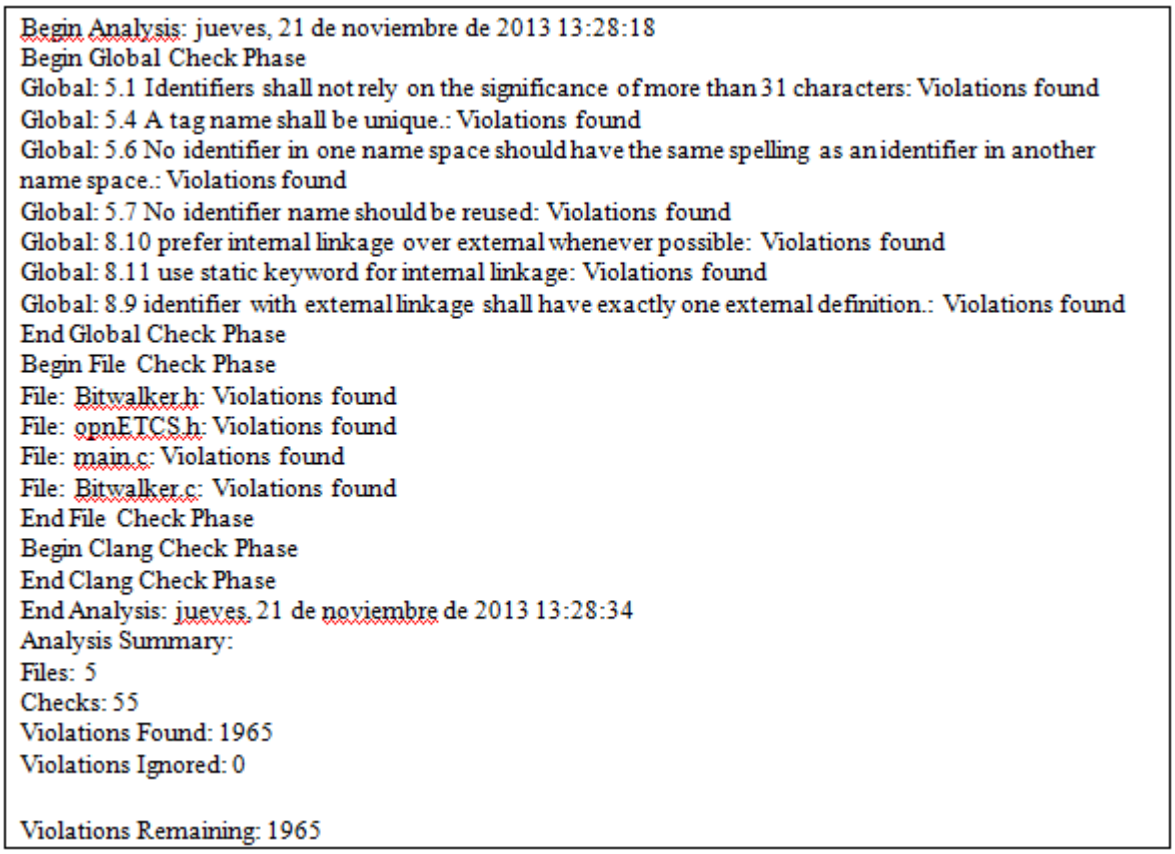


Figure 3. MISRA-C Rules results

The files into the violations are found are listed in the below table.

Table 4. Ubication of MISRA Violations

MISRA Rule	Files
Global 5.1	Bitwalker.c/opnETCS.h/opnETCS_Decoder.h
Global 5.4	opnETCS.h
Global 5.6	Bitwalker.c/Bitwalker.h
Global 5.7	Bitwalker.c/Bitwalker.h/opnETCS.h
Global 8.9	opnETCS_Decoder.h
Global 8.10	main.c
Global 8.11	main.c

3.5 Clang Static Analyzer tool Results

The Clang Static Analyzer is a source code analysis tool that finds bugs in C, C++, and Objective-C programs.

The analyzer is 100% open source and is part of the Clang project. Like the rest of Clang, the analyzer is implemented as a C++ library that can be used by other tools and applications.

With this analysis SQS has checked the following:

Table 5. xxx

core.AdjustedReturnValue	Check to see if the return value of a function call is different than the caller expects (e.g., from calls through function pointers).
core.CallAndMessage	Check for logical errors for function calls and Objective-C message expressions (e.g., uninitialized arguments, null function pointers).
core.DivideZero	Check for division by zero.
core.NonNullParamChecker	Check for null pointers passed as arguments to a function whose arguments are known to be non-null.
core.NullDereference	Check for dereferences of null pointers.
core.StackAddressEscape	Check that addresses to stack memory do not escape the function.
core.UndefinedBinaryOperatorResult	Check for undefined results of binary operators.
core.VLASize	Check for declarations of VLA of undefined or zero size.
core.builtin.BuiltinFunctions	Evaluate compiler built-in functions (e.g., alloca()).
core.builtin.NoReturnFunctions	Evaluate "panic" functions that are known to not return to the caller.
core.uninitialized.ArraySubscript	Check for uninitialized values used as array subscripts.
core.uninitialized.Assign	Check for assigning uninitialized values.
core.uninitialized.Branch	Check for uninitialized values used as branch conditions.
core.uninitialized.CapturedBlockVariable	Check for blocks that capture uninitialized values.
core.uninitialized.UndefReturn	Check for uninitialized values being returned to the caller.
cplusplus.NewDelete	Check for double-free and use-after-free problems involving C++ delete.
deadcode.DeadStores	Check for values stored to variables that are never read afterwards.
osx.API	Check for proper uses of various Apple APIs.
osx.SecKeychainAPI	Check for proper uses of Secure Keychain APIs.
osx.cocoa.AtSync	Check for nil pointers used as mutexes for @synchronized.
osx.cocoa.ClassRelease	Check for sending 'retain', 'release', or 'autorelease' directly to a Class.
osx.cocoa.IncompatibleMethodTypes	Warn about Objective-C method signatures with type incompatibilities.
osx.cocoa.NSAutoreleasePool	Warn for suboptimal uses of NSAutoreleasePool in Objective-C GC mode.
osx.cocoa.NSError	Check usage of NSError** parameters.
osx.cocoa.NilArg	Check for prohibited nil arguments to ObjC method calls.
osx.cocoa.RetainCount	Check for leaks and improper reference count management.
osx.cocoa.SelfInit	Check that 'self' is properly initialized inside an initializer method.
osx.cocoa.UnusedIvars	Warn about private ivars that are never used.
osx.cocoa.VariadicMethodTypes	Check for passing non-Objective-C types to variadic methods that expect only Objective-C types.
osx.coreFoundation.CFError	Check usage of CFErrorRef* parameters.
osx.coreFoundation.CFNumber	Check for proper uses of CFNumberCreate.

Table 5. xxx

osx.coreFoundation.CFRetainRelease	Check for null arguments to CFRetain/CFRelease/CFMakeCollectable.
osx.coreFoundation.containers.OutOfBounds	Checks for index out-of-bounds when using 'CFArray' API.
osx.coreFoundation.containers.PointerSizedValues	Warns if 'CFArray', 'CFDictionary', 'CFSet' are created with non-pointer-size values.
security.FloatLoopCounter	Warn on using a floating point value as a loop counter (CERT: FLP30-C, FLP30-CPP).
security.insecureAPI.UncheckedReturn	Warn on uses of functions whose return values must be always checked.
security.insecureAPI.getpw	Warn on uses of the 'getpw' function.
security.insecureAPI.gets	Warn on uses of the 'gets' function.
security.insecureAPI.mkstemp	Warn when 'mkstemp' is passed fewer than 6 X's in the format string.
security.insecureAPI.mktemp	Warn on uses of the 'mktemp' function.
security.insecureAPI.rand	Warn on uses of the 'rand', 'random', and related functions.
security.insecureAPI.strcpy	Warn on uses of the 'strcpy' and 'strcat' functions.
security.insecureAPI.vfork	Warn on uses of the 'vfork' function.
unix.API	Check calls to various UNIX/Posix functions.
unix.Malloc	Check for memory leaks, double free, and use-after-free problems involving malloc.
unix.MallocSizeof	Check for dubious malloc arguments involving sizeof.
unix.MismatchedDeallocator	Check for mismatched deallocators (e.g. passing a pointer allocating with new to free()).
unix.cstring.BadSizeArg	Check the size argument passed into C string functions for common erroneous patterns.
unix.cstring.NullArg	Check for null pointers being passed as arguments to C string functions.

After run this analysis no violations have been found.

```
Begin Analysis: viernes, 22 de noviembre de 2013 9:23:52
Begin Global Check Phase
End Global Check Phase
Begin File Check Phase
End File Check Phase
Begin Clang Check Phase
End Clang Check Phase
End Analysis: viernes, 22 de noviembre de 2013 9:23:52
Analysis Summary:
Files: 5
Checks: 55
Violations Found: 0
Violations Ignored: 0

Violations Remaining: 0
```

Figure 4. Clang Analysis results

3.6 CPPcheck tool Results

Bitwalker folder has been analyzed statically by CPPcheck tool (Complying with the standard C11). C11 (formerly C1X) is an informal name for ISO/IEC 9899:2011, the current standard for the C programming language. It replaces the previous C standard, informally known as C99. This new version mainly standardizes features that have already been supported by common contemporary compilers, and includes a detailed memory model to better support multiple threads of execution. Due to delayed availability of conforming C99 implementations, C11 makes certain features optional, to make it easier to comply with the core language standard.

The results of the tool are the following:


```
<?xml version="1.0" encoding="UTF-8"?>

-<results version="2"><cppcheck version="1.62"/>-<errors>-<error verbose="Variable &#039;Testwort&#039; is reassigned a
value before the old one has been used." msg="Variable &#039;Testwort&#039; is reassigned a value before the old one has been
used." severity="performance" id="redundantAssignment">

<location line="119" file="\192.168.1.4\DirectorioSQS\Temp\idelatorre\Bitwalker\main.c"/>

<location line="120" file="\192.168.1.4\DirectorioSQS\Temp\idelatorre\Bitwalker\main.c"/>

</error>-<error verbose="Variable &#039;Testwort&#039; is reassigned a value before the old one has been used." msg="Variable
&#039;Testwort&#039; is reassigned a value before the old one has been used." severity="performance"
id="redundantAssignment">

<location line="120" file="\192.168.1.4\DirectorioSQS\Temp\idelatorre\Bitwalker\main.c"/>

<location line="121" file="\192.168.1.4\DirectorioSQS\Temp\idelatorre\Bitwalker\main.c"/>

</error>-<error verbose="Variable &#039;Testwort&#039; is reassigned a value before the old one has been used." msg="Variable
&#039;Testwort&#039; is reassigned a value before the old one has been used." severity="performance"
id="redundantAssignment">

<location line="121" file="\192.168.1.4\DirectorioSQS\Temp\idelatorre\Bitwalker\main.c"/>

<location line="122" file="\192.168.1.4\DirectorioSQS\Temp\idelatorre\Bitwalker\main.c"/>

</error>-<error verbose="Variable &#039;Testwort&#039; is assigned a value that is never used." msg="Variable
&#039;Testwort&#039; is assigned a value that is never used." severity="style" id="unreadVariable">

<location line="122" file="\192.168.1.4\DirectorioSQS\Temp\idelatorre\Bitwalker\main.c"/>

</error></errors></results>
```


Figure 5. cppcheck results

3.7 Results Comparison

4 Conlusions