OETCS/WP4/D4.2.2

openETCS

ITEA2

INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

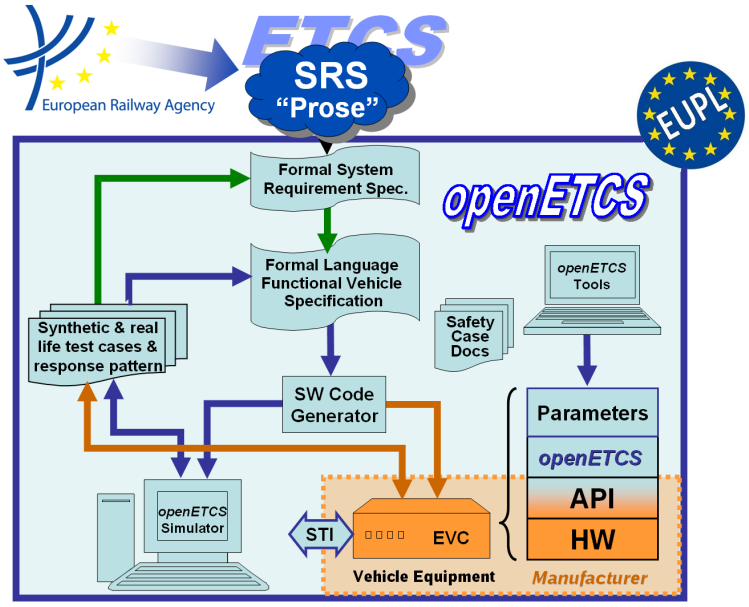
ITEA2 Project
Call 6 11025
2012 – 2015

Work Package 4: “Validation & Verification Strategy”

Preliminary Validation and Verification Report on Implementation/Code

Jens Gerlach and Izaskun de la Torre

March 2015



Funded by:

Federal Ministry of Education and Research

Région de Bruxelles-Capitale

GOBIERNO DE ESPAÑA MINISTERIO DE INDUSTRIA, ENERGÍA Y TURISMO

This page is intentionally left blank

Work Package 4: "Validation & Verification Strategy" **OETCS/WP4/D4.2.2**
March 2015

Preliminary Validation and Verification Report on Implementation/Code

Document approbation

Lead author:	Technical assessor:	Quality assessor:	Project lead:
location / date	location / date	location / date	location / date
signature Jens Gerlach (Fraunhofer FOKUS)	signature Virgile Prevosto (CEA LIST)	signature Abdelnasir Mohamed (AEbt)	signature Klaus-Rüdiger Hase (DB Netz)

Jens Gerlach

Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
jens.gerlach@fokus.fraunhofer.de
www.fokus.fraunhofer.de

Izaskun de la Torre

Software Quality Systems S.A.

Intermediate report

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

List of code examples

List of Corrections

1 Introduction

In this intermediate report we describe the activities to formally verify the correctness of parts of the software developed in the OpenETCS project.

While major parts of the functionality of Subset 026 are modelled in higher-level languages, there is also a substantial part of *supporting* software that is developed in the programming language C.

In this document we report about *preliminary* results on the verification of that C-code. In particular, we report on the use of static analysis methods (including formal methods) on C code that has been developed by the project partner Siemens.

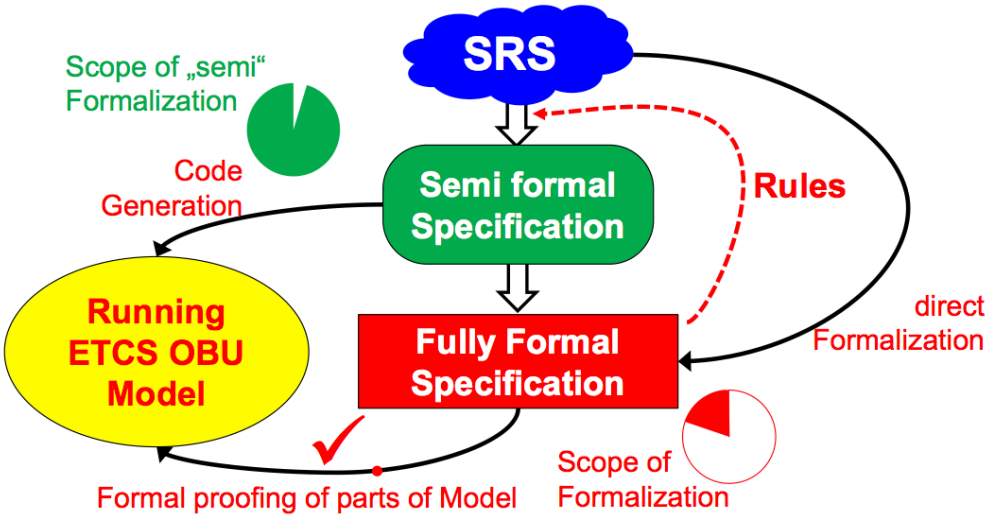


Figure 1.1. Scope of formal methods with in OpenETCS

Figure 1.1 outlines the roles of formal methods within the OpenETCS project. Even a subsystem such as described by *Subset 026* of the ETCS specification is usually too complex to be completely formally specified. Therefore, *semi-formal modelling techniques* and *tests* and *simulations* play a crucial role to verify that the implementation satisfies its specification. However, for clearly defined modules and select system properties, formal methods can well be applied to establish the correctness of an implementation.

Figure 1.2 gives an overview on the software that is in the focus of this report.

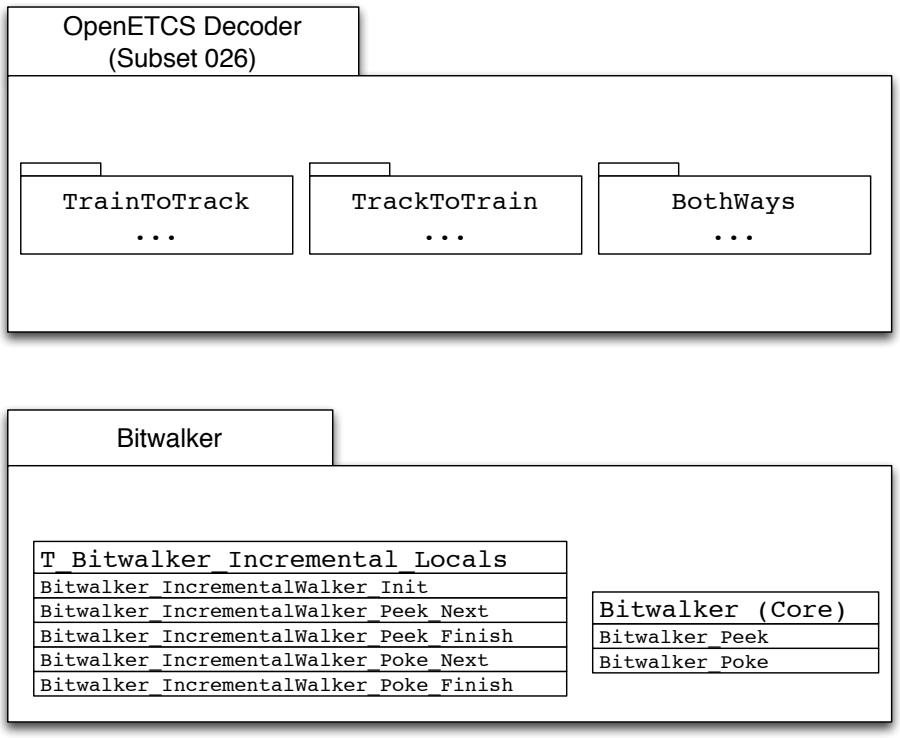


Figure 1.2. The place of `Bitwalker` with the OpenETCS software

The OpenETCS decoder is a large collection of functions dedicated to the reading of ETCS messages. In order to fulfill their task these function rely on the relatively small software package `Bitwalker`. The `Bitwalker` software, as seen by the OpenETCS decoder, is best understood as a "class" with a handful of methods. Note that this class is implemented in C as a `struct` where the methods are implemented as functions. The core functionality of this class, which consists in converting bit sequences to integers and vice versa, depends on two more basic function, namely `Bitwalker_Peek` and `Bitwalker_Poke`.

This software has been analyzed by the OpenETCS project partners SQS (Spain) and Fraunhofer FOKUS (Germany). SQS used several static analysis tools to identify defects and to derive useful metrics. Fraunhofer FOKUS, on the other hand, used the Frama-C tool set, which is developed by the French project partner CEA LIST, in order to *formally verify* various properties of the `Bitwalker`.

These analyses contribute to the ultimate verification goals, which are the following:

1. provide evidence that the Bitwalker software satisfies accepted quality standards
2. develop a formal specification for the Bitwalker software
3. verify that the Bitwalker software satisfies its formal specification
4. show that the Bitwalker software does not raise runtime errors
5. verify that OpenETCS decoder calls the Bitwalker software only according to its specification

We are confident that all these verification goals can be reached. For this preliminary verification report, we provide partial answers to the first four topics. In order to achieve the last goal, more development and verification work is currently conducted by Fraunhofer ESK and Fraunhofer FOKUS.

Structure of this document

Chapter 2 gives a short overview on the Frama-C/WP tool that plays a central role in the verification of the Bitwalker functions. Here we also try to rectify some misunderstandings about formal verification that we have encountered in our work.

In Chapter 3 we analyze the functions `Bitwalker_Peek` and `Bitwalker_Poke` from the Bitwalker core and

1. formally specify the expected functional behavior in the ACSL specification language of Frama-C and
2. report on the formal proof that these C functions do not raise runtime errors when called according to their formal specification, established using the Frama-C verification platform.

So far only a part of Siemens' `Bitwalker` has been formalized and verified. In the process of this work several enhancements for the Frama-C verification platform have been identified and reported to the developers at CEA LIST.

In Chapter 4, we report about the results of SQS' application of a broad range of static analysis tools on the `Bitwalker`. In contrast to Frama-C, these tools cannot exhaustively detect all potential defects of a given kind. Nevertheless, these they are very useful at finding well-known quality deficiencies that might occur in C or C++ software.

In Chapter 5, we draw conclusions from this preliminary work and outline the next steps in our verification efforts.

2 An introduction to formal verification with Frama-C/WP

Frama-C is a platform dedicated to source-code analysis of C software. It has a plug-in architecture and can thus be easily extended to different kinds of analyses. The WP plugin of Frama-C allows one to formally verify that a piece of C code satisfies its specification. This implies, of course, that the user provides a *formal specification* of what the implementation is supposed to do. Frama-C comes with its own specification language ACSL which stands for *ANSI/ISO C Specification Language*. In order to help potential users to master ACSL we discuss in this chapter a very simple C function `abs_int` that implements the computation of the absolute value for objects of type **`int`**.

- In Section 2.1 we will present a straightforward specification of `abs_int`. We discuss the reasons why Frama-C/WP is not able to verify that our implementation satisfies this specification in Section 2.2.
- In Section 2.3 we provide a more precise specification that can be verified by Frama-C/WP. In Section 2.4 we explain how Frama-C supports—by allowing the separation of the specification from the implementation—good software engineering practices.
- Sections 2.5 and 2.6 discuss, respectively, how Frama-C/WP supports *modular verification* and the formal treatment of *side effects*.

2.1 First steps

We will consider the function that computes the absolute value $|x|$ of an integer x. In order to avoid name clashes with the function `abs` in C standard library we use the name `abs_int`.

The mathematical definition of absolute value is very simple

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases} \tag{1}$$

A straightforward implementation of `abs_int` is shown in Listing 2.1.

```
int abs_int(int x)
{
  return (x >= 0) ? x : -x;
}
```

Listing 2.1. An implementation of the absolute value function

In order to demonstrate that this implementation is correct we have to provide a formal specification. Listing 2.2 shows our first attempt for an ACSL specification of `abs_int` that is based on the mathematical definition of the function $x \mapsto |x|$ in Equation 1.

```
/*@
    ensures 0 <= x ==> \result ==  x;
    ensures 0 >  x ==> \result == -x;
*/
int abs_int(int x)
{
  return (x >= 0) ? x : -x;
}
```

Listing 2.2. A first attempt to formally specify `abs_int`

The first thing to note is that ACSL specifications—or *contracts*—are placed in special C comments (they start with `/*@`). Thus, they do not interfere with the executable code. The **`ensures`** clause in the specification expresses *postconditions*, that is, properties that should be guaranteed *after* the execution of `abs_int`. The ACSL reserved word `\result` is used to refer to the return value of a C function. Note that we use the usual C operators == and <= to express equalities and inequalities in the specification. There is also an additional operator ==> which expresses logical implication.

2.2 Why can Frama-C/WP not verify such a simple function?

Although the specification and implementation in Listing 2.2 look perfectly right, Frama-C/WP cannot verify that the implementation actually satisfies its specification.

The reason becomes clear if we look at some actual return values of `abs_int`. Listing 2.3 shows our test code whose output is listed in Table 2.1.

```c
#include <stdio.h>
#include <limits.h>

extern int abs_int(int);

void print_abs(int x)
{
  printf("%12d\t\t%12d\n", x, abs_int(x));
}

int main()
{
  printf("\n");
  print_abs(0);

  printf("\n");
  print_abs(1);
  print_abs(10);
  print_abs(INT_MAX);

  printf("\n");
  print_abs(-1);
  print_abs(-10);
  print_abs(INT_MIN);
}
```

Listing 2.3. Some simple test cases for `abs_int`

x	abs_int(x)	Remark
0	0	✓
1	1	✓
10	10	✓
2147483647	2147483647	✓
-1	1	✓
-10	10	✓
-2147483648	-2147483648	↯

Table 2.1. Test results for `abs_int`

The offending value is in the last line of Table 2.1 which basically states that `abs_int(INT_MIN)` equals `INT_MIN` whereas it should equal `-INT_MIN`. The problem is that the type **`int`** only present a finite subset of the (mathematical) integers. Many computers use a two's-complement representation of integers which covers the range $[-2^{31} \ldots 2^{31} - 1]$ on a 32-bit machine. On such a machine `-INT_MIN` cannot be represented by a value of the type **`int`**.

In a specification, Frama-C/WP interprets integers as mathematical entities. Consequently, there is no such thing as an *arithmetic overflow* when adding or multiplying them. In other words, Frama-C/WP is perfectly right not being able to verify that `abs_int` satisfies the contract in Listing 2.2. Indeed, the implementation does not respect the given specification.

2.3 Sharpening the contract of `abs_int`

It is of course well known that the operation `-x` can overflow and it is the fact that Frama-C/WP can detect such overflows that helps to prevent incorrect verification results.

The GNU Standard C Library clearly states that the absolute value of `INT_MIN` is undefined.[1] Under OSX, the manual page of `abs` mentions under the field of "Bugs":

```
The absolute value of the most negative integer remains negative.
```

Thus, our formal specification should exclude the value `INT_MIN` from the set of admissible value to which `abs_int` can be applied. In ACSL, we can use the **`requires`** clause to express *preconditions* of a function. Listing 2.4 shows an extended contract of `abs_int` that takes the limitations of the type **`int`** into account.

```
#include <limits.h>

/*@
    requires x > INT_MIN;

    ensures 0 <= x ==> \result ==  x;
    ensures 0 >  x ==> \result == -x;
*/
int abs_int(int x)
{
  return (x >= 0) ? x : -x;
}
```

Listing 2.4. Taking integer overflows into account

Frama-C/WP is now capable to verify that the implementation of `abs_int` satisfies the specification of Listing 2.4.

There is an important lesson that can be learned here:

> Sometimes developers provide source code and imagine that a tool like Frama-C/WP can verify the correctness of their implementation. In order to fulfill its task, however, Frama-C/WP needs an ACSL specification. Such a specification—which must be based on a reasonably precise description of the admissible inputs and expected behavior—has to come from the *requirements* of the software and is not magically discovered from the source code by Frama-C/WP. The code does what it does. In order to verify that the code does what someone expects, these expectations must be clearly expressed, that is, they must be formally specified.

Of course, it might not always be the goal to verify the complete functionality of a piece of software. Sometimes, it is enough to ensure that individual software components cause no runtime errors, that is, arithmetic overflows or invalid pointer accesses. Frama-C/WP can also

[1] See `http://www.gnu.org/software/libc/manual/html_node/Absolute-Value.html`

be used in this situation. Under the terms of the following minimal specification in Listing 2.5, Frama-C/WP can verify that no runtime error will occur.

```
#include <limits.h>

/*@
    requires x != INT_MIN;
*/
int abs_int(int x)
{
  return (x >= 0) ? x : -x;
}
```

Listing 2.5. Minimal contract to ensure the absence of runtime errors in `abs_int`

2.4 Separating specification and implementation

Before we continue exploring more advanced specification and verification capabilities of Frama-C/WP we turn to a simple software engineering question.

It is common practice to put function prototypes into "`.h`" files and keep the implementation in files ending in "`.c`". Frama-C/WP supports this separation of specification and implementation. Listing 2.6 shows the file `abs2.h` which contains a declaration of `abs_int` together with an attached ACSL specification.

```
#include <limits.h>

/*@
    requires x > INT_MIN;

    ensures 0 <= x ==> \result ==  x;
    ensures 0 >  x ==> \result == -x;
*/
int abs_int(int x);
```

Listing 2.6. Specifying a function prototype in a header file

Listing 2.7 shows the specification of `abs_int` in a `.c` file. Note that the file `abs2.h` with the specification is included by this file. Frama-C/WP can verify that this implementation satisfies the contract in Listing 2.6.

```
#include "abs2.h"

int abs_int(int x)
{
  return (x >= 0) ? x : -x;
}
```

Listing 2.7. Implementation at a different location than the specification

We remark, that the definition of a very small function like `abs_int` would normally be placed in a header file so that a compiler can inline the function definition at the call site.

2.5 Modular verification

We now look at a simple example in which our function `abs_int` is used. More precisely, we include in Listing 2.8 the header file from Listing 2.6 which contains an ACSL specification of `abs_int`.

```
#include "abs2.h"

void use_1()
{
  int a = abs_int(3);
  int b = abs_int(-1);
  int c = abs_int(INT_MAX);
  int d = abs_int(INT_MIN);

  // ...
}
```

Listing 2.8. A simple example of modular verification

When Frama-C/WP tries to verify the code in Listing 2.8, then it actually tries to establish whether at each program location where it is called the *preconditions* of `abs_int` are satisfied. Based on the specification of `abs_int`, Frama-C/WP can indeed verify that for the first three calls the preconditions are fulfilled. For the last call this verification fails because the value `INT_MIN` is explicitly excluded by the specification in Listing 2.6.

Note that the *implementation* of `abs_int` does not play any role in determining whether it is safe to call the function in a particular context. This is what we call *modular verification*: a function can be verified in isolation whereas code that calls the function only uses the function contract.

This also means that in a situation as in Listing 2.9, where nothing is known about the argument of `abs_int`, Frama-C/WP cannot establish that the precondition of `abs_int` is satisfied or, in other words, that `x > INT_MIN` holds.

```
#include "abs2.h"

void use_2(int x)
{
  int a = abs_int(x);

  // ...
}
```

Listing 2.9. Another example of modular verification

If, on the other hand, we have precise information on the arguments at call site, then Frama-C/WP can exploit the specification of `abs_int` in order derive some interesting properties. As an example, we consider the code fragment in Listing 2.10. Here, Frama-C/WP can verify that the assertion after the call of `abs_int` is correct.

```
#include "abs2.h"

/*@
    requires (10 <= x < 100) || (-200 < x < -50);
*/
void use_3(int x)
{
  int a = abs_int(x);
  //@ assert 10 <= a < 200;

  // ...
}
```

Listing 2.10. A more complex example of modular verification

Note that this assertion is a *static* one, that is, it is an ACSL annotation that resides inside a comment and does not affect the execution of the code in Listing 2.10. Also note that unlike to C code, *relation chains* can be used both in function contracts and assertions.

2.6 Dealing with side effects

Listing 2.11 shows an implementation of `abs_int` that writes as a side effect the argument `x` to a global variable `a`. A natural question is to ask whether this implementation with a side effect also satisfies the specification.

```
#include <limits.h>

extern int a;

/*@
    requires x > INT_MIN;

    ensures 0 <= x ==> \result ==  x;
    ensures 0 >  x ==> \result == -x;
*/
int abs_int(int x)
{
  a = x; // Is this side effect covered by the specification?
  return (x >= 0) ? x : -x;
}
```

Listing 2.11. An implementation with side effects

Before we answer this question we consider various uses for side effects. There are of course legitimate uses for side effects. The assignment to a memory location outside the scope of the function might be meaningful because an error condition is reported or because some data are logged as in Listing 2.12.

If Frama-C/WP attempts to verify the code in Listing 2.12, then it issues the following warning:

```
Neither code nor specification for function logging,
generating default assigns from the prototype
```

```
#include <limits.h>

extern void logging(int);

/*@
    requires x > INT_MIN;

    ensures 0 <= x ==> \result ==  x;
    ensures 0 >  x ==> \result == -x;
*/
int abs_int(int x)
{
  logging(x);
  return (x >= 0) ? x : -x;
}
```

Listing 2.12. Calling a logging function from `abs_int`

Thus, it points out that the called function `logging` should have a proper specification that clearly indicates its side effects.

There are, on the other hand, also good reasons to minimize or even forbid side effects:

- Imagine a malicious password checking function that writes the password to a global variable.
- Another reason is that side effects can make it harder to understand what the real consequences of a function call are. In particular, one must be concerned about unintended consequences that are caused by side effects The norm IEC 61508 therefore requests in the context of software module testing and integration testing:

  > To show that all software modules, elements and subsystems interact correctly to perform their intended function and do not perform unintended functions (see also. [1, §7.4.7.2,§7.7.2.9])

  Of course, it is quite difficult to ensure by testing alone that something does *not* happen.

To come back to our question about Listing 2.11 it is important to understand that Frama-C/WP verifies that the implementation shown there satisfies the specification.

If one wishes to forbid that a function changes global variables one can use an **`assigns`** `\nothing` clause as shown in Listing 2.13. Frama-C/WP will then point out that this implementation prevents the verification of the assigns clause.

```
#include <limits.h>

extern int a;

/*@
    requires x > INT_MIN;

    assigns \nothing; // forbid any side effects

    ensures 0 <= x ==> \result ==  x;
    ensures 0 >  x ==> \result == -x;
*/
int abs_int(int x)
{
  a = x; // now illegal
  return (x >= 0) ? x : -x;
}
```

Listing 2.13. Specifying the absence of side effects

Of course, an all-or-nothing-approach to side effects is not very helpful for the verification of real-life software. Listing 2.14 shows how the **assigns** clause of a specification can name the exact memory location that the function is allowed to modify.

```
// Side effects can be controlled on an individual basis.

#include <limits.h>

extern int a;

/*@
    requires x > INT_MIN;

    assigns a; // allow assignment to a (but only to a).

    ensures 0 <= x ==> \result ==  x;
    ensures 0 >  x ==> \result == -x;
*/
int abs_int(int x)
{
  a = x;
  return (x >= 0) ? x : -x;
}
```

Listing 2.14. Finer control of side effects

Note however that **assigns** a does not imply that a write to a necessarily occurs during the execution of abs. On the other hand, any other memory location must stay unchanged between the initial state and the final state of abs.

3 Formal verification of the `Bitwalker` core functionality

In this chapter we describe in some detail our efforts on formally verifying the `Bitwalker` using the verification tool Frama-C/WP. 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.

In Chapter 2 we had given an introduction into the most elementary features of Frama-C/WP. In this chapter we use slightly more complex features of Frama-C/WP in order to verify key aspects of the `Bitwalker` functions `Bitwalker_Peek` and `Bitwalker_Poke`. In particular, we formally specify and (almost completely) verify the main operational modes of these functions. In particular, we show that no runtime errors will occur if the preconditions of the operational modes are satisfied.

After shortly describing our verification method in Section 3.1, we discuss in Section 3.2 the use of Frama-C/WP to detect *potential* runtime errors in `Bitwalker_Peek` and `Bitwalker_Poke`. In Section 3.3 we present an *informal specification* for both functions. This informal specification, together with the knowledge about potential run time errors, serves two main purposes.

1. It allows us to quickly write tests for the `Bitwalker` functions (see Section 3.4).
2. It serves as a basis for the *formal* specification in the ACSL specification language of Frama-C/WP (see Section 3.5).

In Section 3.5 we also discuss several modifications of the source code that simplify the verification with Frama-C/WP.

Since our formal specification of `Bitwalker` is not yet complete, we present in Section 3.6 *preliminary* verification results that we achieved with Frama-C/WP. Finally, we give in Section 3.7 an overview about the issues that are still open.

3.1 Verification method

In Chapter 2 we have given an introduction to some of the capabilities of Frama-C/WP. In this section we give a shorter, higher-level presentation of our verification approach.

We use *deductive verification* in order to formally prove that a function satisfies its specification. The foundations for deductive verification are axiomatic semantics as formulated by Hoare [2]. Figure 3.1 shows the method with the involved verification tools.

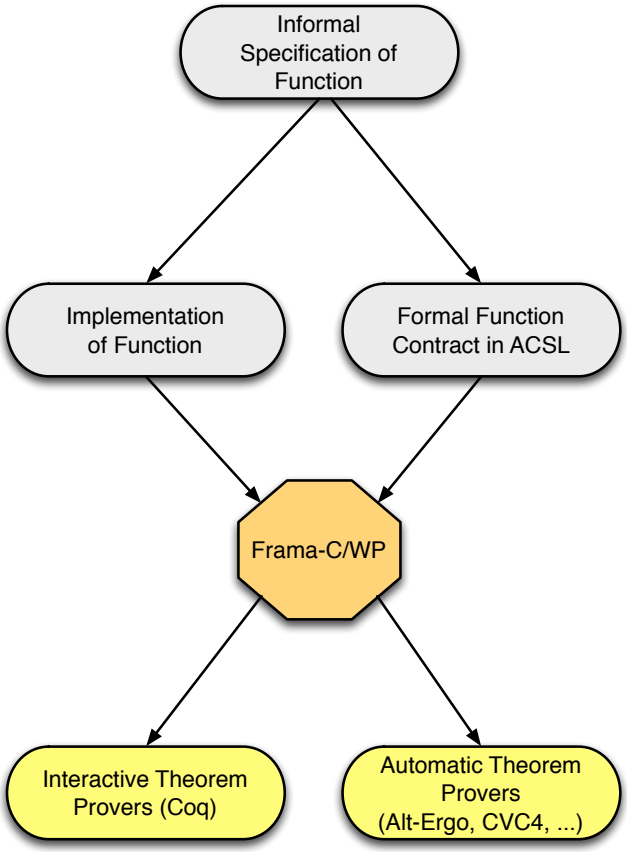


Figure 3.1. Deductive verification of C code with Frama-C/WP.

Starting point is an informal specification of a function with which in mind an implementation is written. This informal specification is then formalized using the specification language ACSL (ANSI/ISO-C Specification Language) [3] that comes with Frama-C and is a formal language to express behavioral properties of C programs. 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.

ACSL is the specification language associated with the verification platform Frama-C [4] which we use along with its plug-in Frama-C/WP [5]. Within Frama-C, the WP plug-in supports the deductive verification of C programs that have been annotated with ACSL. Frama-C/WP generates verification conditions which are submitted to automatic or interactive theorem provers. If each verification condition is discharged by at least one prover, then the implementation of the function satisfies its contract.

Figure 3.1 shows that we apply the automatic theorem provers Alt-Ergo [6] and CVC4 [7] and then, if necessary, apply the interactive theorem prover Coq [8] for remaining unproven

conditions. Moreover, unproven conditions motivate to give some extra information in the form of axioms, lemmas, or assertions in ACSL, since these can ease the search of a proof. One needs to be careful with axioms because they can yield contradictions and thus make the proof system unsound.

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

We received the source code only with a high-level description of what the `Bitwalker` is supposed to do. In particular, no sufficient information about error conditions were provided. On such a basis it is, as pointed out on Page 8, not possible to write meaningful test cases, let alone to formally verify the functionality of the bitwalker functions.

In a first step, we therefore had to inspect the source code and derive from it an *informal specification*. This informal specification is to be understood as a requirements document for the bitwalker functions as it should have been available to both the programmer and the verifier in advance.

There are several problems with this approach:

- The verifier could make an error while analyzing the source code and end up with a wrong specification.
- If the verifier's analysis is correct, there could still be an error in the implementation which would then be present also in the specification. In any case, only the trivial claim "the code works as implemented" can be verified.

In order to avoid these problems we submitted our informal specification for review by the project partner Siemens.

3.2 A first look on `Bitwalker_Peek` and `Bitwalker_Poke`

In this section, we analyze the implementations of `Bitwalker_Peek` and `Bitwalker_Poke`. The goal is to devise a more precise specification than was originally provided. Of course, a specification derived from the source code by the verifier must be subject to a review of the domain experts.

At this point we are already using Frama-C/WP in order to identify potential run time errors in the source code.

3.2.1 Analyzing `Bitwalker_Peek`

Listing 3.1 shows the original implementation of `Bitwalker_Peek`.

```
#include "Bitwalker.h"

uint64_t Bitwalker_Peek(unsigned int Startposition,
                        unsigned int Length,
                        uint8_t Bitstream[],
                        unsigned int BitstreamSizeInBytes)
{
  if (((Startposition + Length - 1) >> 3) >= BitstreamSizeInBytes)
    return 0; // error: index out of range

  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 3.1. Original implementation of `Bitwalker_Peek`

Here are some remarks on this implementation.

- The implementation extensively uses bit operations. This is of course largely a matter of taste. Nevertheless, it is questionable whether representing a division of an index `i` by 8 as `i >> 3` is better than writing it as `i/8`.
- The argument `Bitstream` represents an array that is only read. It is good programming practice to qualify such arguments as `const`. Likewise, `CurrentValue` should be declared as `const`.
- The cast of `CurrentValue != 0` to `uint8_t` is unnecessary for the following reasons:
  - The result of expression `CurrentValue != 0` is of type `int` and has either the value 1 or 0.

– According to the "usual arithmetic conversions"[2] this value will be promoted to the type of `retval << 1` which is **`uint64_t`**.

Thus, the cast to **`uint8_t`** is pointless and removing it increases the clarity of the code.

At one point, an alternative to the implementation of `Bitwalker_Peek` in Listing 3.1 was suggested. This alternative implementation, which is shown in Listing 3.2 attempts to limit the use of bit operations to a minimum.

```
uint64_t Bitwalker_Peek (unsigned int Startposition,
                         unsigned int Length,
                         uint8_t Bitstream[],
                         unsigned int BitstreamSizeInBytes)
{
  uint64_t retval = 0;
  for (unsigned int i = Startposition +
         BitstreamSizeInBytes*!((Startposition + Length) <=
         BitstreamSizeInBytes*8);
         i < Startposition + Length; i++)
    retval = (retval*2) +
             (uint8_t)((uint8_t)!!(Bitstream[i/8] &
                BitwalkerBitMaskTable[i%8]));
  return retval;
}
```

Listing 3.2. An alternative implementation of `Bitwalker_Peek`

Interestingly, this implementation also employs unnecessary casts to **`uint8_t`**. However, the real problem with this alternative implementation is that it produces different results: Calling `Bitwalker_Peek` from Listing 3.1 with the arguments

```
Startposition = 8
Length = 32
Bitstream[] = {254, 7, 13, 9}
BitstreamSizeInBytes = 4
```

produces 0 whereas the implementation from Listing 3.2 returns 118294784. Apparently, even `Bitwalker_Peek` is not so simple that its functionality can be unambiguously understood just by reading the code.

[2] This is indeed the heading of Section 6.3.1.8 of the C standard.

Figure 3.2 shows a representation of `Bitwalker_Peek`, normalized and enhanced with static ACSL assertions by Frama-C/WP. These assertions can be generated by Frama-C for all operations where runtime errors, that is, illegal pointer accesses or arithmetic overflows, can occur. Green bullets indicate potential runtime errors where Frama-C/WP can verify that they will *not* occur.

```
uint64_t Bitwalker_Peek(unsigned int Startposition, unsigned int Length,
                        uint8_t *Bitstream, unsigned int BitstreamSizeInBytes)
{
  uint64_t __retres;
  uint64_t retval;
  unsigned int i;
  /*@ assert
      rte: unsigned_overflow:
        0 <= (unsigned int)(Startposition+Length)-(unsigned int)1;
  */
  /*@ assert rte: unsigned_overflow: 0 <= Startposition+Length; */
  /*@ assert rte: unsigned_overflow: Startposition+Length <= 4294967295; */
  if (((Startposition + Length) - (unsigned int)1) >> 3 >= BitstreamSizeInBytes) {
    __retres = (unsigned long long)0;
    goto return_label;
  }
  retval = (unsigned long long)0;
  i = Startposition;
  while (1) {
    /*@ assert rte: unsigned_overflow: 0 <= Startposition+Length; */
    /*@ assert rte: unsigned_overflow: Startposition+Length <= 4294967295; */
    if (! (i < Startposition + Length)) {
      break;
    }
    {
      uint8_t CurrentValue;
      /*@ assert rte: mem_access: \valid_read(Bitstream+(unsigned int)(i>>3));
      */
      /*@ assert rte: index_bound: (unsigned int)(i&(unsigned int)0x07) < 8;
      */
      CurrentValue = (unsigned char)((int)*(Bitstream + (i >> 3)) & (int)BitwalkerBitMaskTable[
                                     i & (unsigned int)0x07]);
      /*@ assert
          rte: unsigned_overflow:
            0 <=
            (unsigned long long)(retval<<1)+(unsigned long long)((unsigned char)
                                                                 ((int)
                                                                   ((int)CurrentValue!=0)));
      */
      /*@ assert
          rte: unsigned_overflow:
            (unsigned long long)(retval<<1)+(unsigned long long)((unsigned char)
                                                                 ((int)
                                                                   ((int)CurrentValue!=0)))
            <= 18446744073709551615;
      */
      retval = (retval << 1) + (uint64_t)((unsigned char)((int)CurrentValue != 0));
    }
    /*@ assert rte: unsigned_overflow: i+1 <= 4294967295; */
    i ++;
  }
  __retres = retval;
  return_label: return __retres;
}
```

Figure 3.2. Potential runtime errors in `Bitwalker_Peek`

The remaining potential runtime errors, marked yellow, are related to the facts that at this point Frama-C/WP

- cannot exclude that `Length` can be greater then 64
- has to assume that `Startposition + Length` may overflow
- has no guarantee that `BitstreamSizeInBytes` is the length of the array starting at the address `Bitstream`

3.2.2 Analyzing `Bitwalker_Poke`

Listing 3.3 shows the original implementation of `Bitwalker_Poke`.

```
#include "Bitwalker.h"

int Bitwalker_Poke (unsigned int Startposition,
                    unsigned int Length,
                    uint8_t Bitstream[],
                    unsigned int BitstreamSizeInBytes,
                    uint64_t Value)
{
  // plausibility check: is last byte in range
  if (((Startposition + Length - 1) >> 3) >= BitstreamSizeInBytes)
    return -1;  // error: index out of range

  // plausibility check: is value in range
  uint64_t MaxValue = (((uint64_t)0x01) << Length) - 1;

  if (MaxValue < Value)
    return -2;  // error: value to big for bit field

  // Everything ok, we can iterate bitwise from left to right
  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 3.3. Original implementation of `Bitwalker_Poke`

Clearly visible in the code are various error conditions that are checked by `Bitwalker_Poke`. No specifications for these error conditions have been provided.

Figure 3.3 shows the normalized representation of `Bitwalker_Poke` with ACSL assertions that indicate potential runtime errors.

```
int Bitwalker_Poke(unsigned int Startposition, unsigned int Length,
                   uint8_t *Bitstream, unsigned int BitstreamSizeInBytes,
                   uint64_t Value)
{
  int __retres;
  uint64_t MaxValue;
  int i;
  /*@ assert
      rte: unsigned_overflow:
        0 <= (unsigned int)(Startposition+Length)-(unsigned int)1;
  */
  /*@ assert rte: unsigned_overflow: 0 <= Startposition+Length; */
  /*@ assert rte: unsigned_overflow: Startposition+Length <= 4294967295; */
  if (((Startposition + Length) - (unsigned int)1) >> 3 >= BitstreamSizeInBytes) {
    __retres = -1;
    goto return_label;
  }
  /*@ assert
      rte: unsigned_overflow:
        0 <=
        (unsigned long long)((unsigned long long)0x01<<Length)-(unsigned long long)1;
  */
  /*@ assert rte: shift: 0 <= Length && Length < 64; */
  MaxValue = ((unsigned long long)0x01 << Length) - (unsigned long long)1;
  if (MaxValue < Value) {
    __retres = -2;
    goto return_label;
  }
  /*@ assert
      rte: unsigned_overflow:
        0 <= (unsigned int)(Startposition+Length)-(unsigned int)1;
  */
  /*@ assert rte: unsigned_overflow: 0 <= Startposition+Length; */
  /*@ assert rte: unsigned_overflow: Startposition+Length <= 4294967295; */
  i = (int)((Startposition + Length) - (unsigned int)1);
  while (i >= (int)Startposition) {
    if ((Value & (unsigned long long)0x01) == (unsigned long long)0) {
      /*@ assert rte: mem_access: \valid(Bitstream+(int)(i>>3)); */
      /*@ assert rte: shift: 0 <= i; */
      /*@ assert rte: mem_access: \valid_read(Bitstream+(int)(i>>3)); */
      /*@ assert rte: index_bound: 0 <= (int)(i&0x07); */
      /*@ assert rte: index_bound: (int)(i&0x07) < 8; */
      *(Bitstream + (i >> 3)) = (unsigned char)((int)*(Bitstream + (i >> 3)) & ~ ((int)BitwalkerBitMaskTable[
                                                i & 0x07]));
    }
    else {
      /*@ assert rte: mem_access: \valid(Bitstream+(int)(i>>3)); */
      /*@ assert rte: shift: 0 <= i; */
      /*@ assert rte: mem_access: \valid_read(Bitstream+(int)(i>>3)); */
      /*@ assert rte: index_bound: 0 <= (int)(i&0x07); */
      /*@ assert rte: index_bound: (int)(i&0x07) < 8; */
      *(Bitstream + (i >> 3)) = (unsigned char)((int)*(Bitstream + (i >> 3)) | (int)BitwalkerBitMaskTable[
                                                i & 0x07]);
    }
    Value >>= 1;
    /*@ assert rte: signed_overflow: -2147483648 <= i-1; */
    i --;
  }
  __retres = 0;
  return_label: return __retres;
}
```

Figure 3.3. Potential runtime errors in `Bitwalker_Peek`

Similarly to the potential runtime errors of `Bitwalker_Peek`, Frama-C/WP is faced with the problem that it

- cannot exclude that `Length` is greater than 64
- has to assume that `Startposition + Length` may overflow
- has no guarantee that `BitstreamSizeInBytes` is the length of the array starting at the address `Bitstream`

3.3 Informal specifications

In the following, we present the informal specification that Fraunhofer FOKUS derived from analysing the implementation of `Bitwalker`.

3.3.1 Basic concepts

First we introduce various terms that we will use in our informal specifications. In particular, we distinguish between *bit streams* and *bit sequences*.

- A *bit stream* is an array containing elements of type `uint8_t`.
- If `a` is the starting address of a bit stream and if all pointers `a+[0..n-1]` are *valid* in the sense of the C standard (cf. [9, § 6.5.3.2(4)]), then we refer to `a` as a *valid bit stream of length* n.
- A bit stream of length n contains $8n$ bits.
- A bit stream can be indexed both by array indices and *bit indices*.

  Figure 3.4 shows the difference between array indices (bottom row) and bit indices (top row) in a bit stream. The two bit indices, 0 and 14, mark bit positions in the first and second array element, respectively.

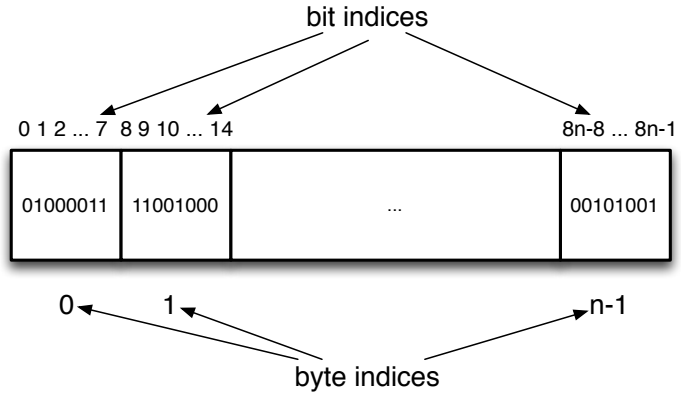


Figure 3.4. Byte indices and bit indices in a bit stream

- The C programming language neither provides a type *bit* nor does it support random access to the bits of a bit stream. In order to access the i-th bit of a bit sequence one typically has to first access the byte with index $j = i/8$ and then access the bit $k = i\%8$ within this byte. Note that in Figure 3.4 bytes and bits are indexed in increasing order, starting from the *left*. In big-endian mode, however, bits are indexed from the *right*. For example, to access the k-th bit (from the left) of a byte `a` one can shift this byte to the right by $7 - k$ bits and then extract the now rightmost bit by performing a bit-wise *and* with the value 1

```
(a >> (7-k)) & 1          // get the k-th left-most bit of a
```

- A *bit sequence* is a consecutive sequence of bits within a bit stream as represented in Figure 3.5.

  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.

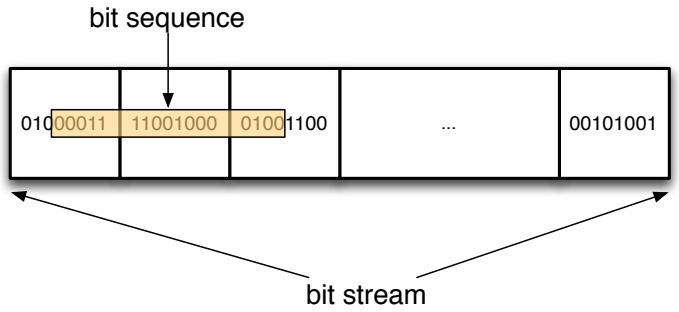


Figure 3.5. A bit sequence within a bit stream

- A bit sequence that starts at bit index p and has length $l \geq 0$ is considered *valid* (with respect to a bit stream of length n) if the following conditions are satisfied

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

  Note that only the bits with indices $p \leq i < p + l$ are to be accessed but not the bit with index $p + l$.

We assume that the C-types `unsigned int` and `int`, which are used in the implementation to represent indices, counting and error codes, have a width of 32 bits. We point this out here because we conducted the verification on a platform with these characteristics.

As an aside, MISRA-C discourages the use of "generic" integer types such as `int` and `unsigned int` and recommends the use of integer types whose names contain the exact width.

3.3.2 Informal specification of `Bitwalker_Peek`

Now we specify `Bitwalker_Peek` based on the introduced auxiliary concepts. The function `Bitwalker_Peek` reads a bit sequence from a bit stream and converts it to a 64-bit long integer.

Its function signature reads as follows:

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

Arguments

The arguments of `Bitwalker_Peek` 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.

Preconditions

The following preconditions shall hold for the function arguments. Note that additional constraints are implicitly expressed by the use of *unsigned* integer types.

- `Bitstream` is a valid array of length `BitstreamSizeInBytes`
- `Length` $\leq$ `64` and
- `Startposition` $\leq$ `UINT_MAX - Length`. This condition expresses that no arithmetic overflows shall occur when evaluating `Startposition + Length`.

Description

As mentioned, the function `Bitwalker_Peek` reads a bit sequence from a bit stream and converts it to a 64-bit unsigned integer.

For a bit sequence $(b_0, b_1, \ldots, b_{n-1})$ the function `Bitwalker_Peek` returns the sum

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

Note that is a higher-level description than what is done in the source code. There is, in our opinion, not much point to reflect all of the low-level bit operations into the specification if a clearer description is at hand.

If the bit sequence is not valid, then `Bitwalker_Peek` shall return 0 according to the Siemens high-level description. We were wondering why the implementation maps both an illegal input and a legal one to the same output. The code providers argued along the lines that this error condition was not considered important enough to be properly reported. One can interpret this design decision as an attempt to increase the robustness of the function against illegal values. In general, we recommend to explicitly describe all error conditions and to devise a consistent error detection and error recovery strategy.

3.3.3 Informal specification of `Bitwalker_Poke`

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

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);
```

Arguments

The arguments have the following purpose:

- `Startposition` is the bit index in the bit stream where the bit sequence shall start.
- `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 to be converted into the bit sequence.

Preconditions

The following conditions shall hold for the function arguments:

- `Bitstream` is a valid array of length `BitstreamSizeInBytes`
- `Startposition + Length` is less than or equal to `UINT_MAX`.

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

Description

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

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

$$\sum_{i=0}^{n-1} b_i \cdot 2^{(n-1)-i} = x. \tag{3}$$

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 not valid, then `Bitwalker_Poke` returns -1.
- If the bit sequence is valid, then there are two cases:
  - If x is greater or equal than $2^{\texttt{Length}}$, then x cannot be represented as bit sequence $(b_0, b_1, \ldots, b_{\texttt{Length}-1})$. `Bitwalker_Poke` returns then -2.
  - If x is less the $2^{\texttt{Length}}$, 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.

3.4 Relationship with `boost::dynamic_bitset`

While analyzing `Bitwalker_Peek` and `Bitwalker_Poke` we also had a look at the C++ class `boost::dynamic_bitset`. This class, which is part of the Boost libraries, provides a higher-level and easier to use interface to bit sequences than is possible in C.[3]

Specifically, we have used `boost::dynamic_bitset` with the following typedefs

```
typedef std::vector<uint8_t>              Bytestream;

typedef boost::dynamic_bitset<uint8_t>    Bitstream;
```

to represent arrays of sequences of bytes and bits, respectively. An object of type `Bitstream` can be initialized with an object of type `Bytestream`. The type `Bitstream` offers random access to its stored bits. In addition, it allows to

- compute the unsigned value represented in the bit stream by calling the method `to_ulong()`, thereby representing the functionality of `Bitwalker_Peek`
- create a bit stream from an unsigned integer value by a special constructor, thus representing the functionality of `Bitwalker_Poke`

While systematic testing the `Bitwalker` was not our main objective it proved useful for the following reasons.

- It helped us formulating the formal specifications of `Bitwalker_Peek` and `Bitwalker_Poke`.
- It allowed us to quickly detect that the alternative implementation of `Bitwalker_Peek` in Listing 3.2 is not equivalent to the original implementation in Listing 3.1.

If `boost:dynamic_bitset` did not, as its name already suggests, rely on dynamic memory allocation, it would be a more suitable candidate to represent the `Bitwalker` functionality, than the hand-crafted and somewhat contrived code at hand.

[3] See `http://www.boost.org/doc/libs/1_55_0/libs/dynamic_bitset/dynamic_bitset.html`

3.5 Formal specification with ACSL

In this section, we discuss formal contracts for `Bitwalker_Peek` and `Bitwalker_Poke`. The contracts are written in ACSL. Note that they do not provide a full formal specification of the functionality of the respective functions. As of now they describe the main operation modes and are aimed at showing that no runtime errors can occur if the functions are called in a context where their preconditions are satisfied.

3.5.1 Formal specification of `Bitwalker_Peek`

Listing 3.4 shows an ACSL contract with the main operation modes of `Bitwalker_Peek`. We have labeled various clauses of the contract (by user-defined identifiers between initial keyword and colon, e.g. "`readable_bitstream`"). This feature of ACSL allows us to refer to them more easily. Note also that we sometimes use shorter names than in the original implementation.

```
#include "Bitwalker.h"

/*@
  requires readable_bitstream:
           \valid_read(Bitstream + (0..BitstreamSize-1));
  requires valid_length: 0 <= Length <= 64;
  requires no_overflow_1: Start + Length <= UINT_MAX;
  requires no_overflow_2: 8 * BitstreamSize <= UINT_MAX;

  assigns \nothing;

  behavior  invalid_bit_sequence:
    assumes (Start + Length)  > 8 * BitstreamSize;
    assigns \nothing;
    ensures \result == 0;

  behavior  normal_case:
    assumes (Start + Length) <= 8 * BitstreamSize;
    assigns \nothing;
    ensures no_overflow_on_result: \result < (1 << Length);

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

Listing 3.4. Formal specification of `Bitwalker_Peek` in ACSL

The structure of this contract is as follows:

Default behavior

- The property `readable_bitstream` uses the built-in ACSL predicate `\valid_read`. This expresses that all addresses in the range `Bitstream[0..BitstreamSize-1]` can be safely dereferenced for *reading*, but not necessarily for writing.
- The property `valid_length` expresses the requirement that only bit sequences with a length not larger than 64 are to be read.

- The two `overflow` properties request that no arithmetic overflow shall occur for the expressions `Start + Length` and `8 * BitstreamSize`. In ACSL, an unbounded-precision type **`integer`** is used as default, such that a contract formula can't cause an overflow by itself.

  Given the operational context of the `Bitwalker` these overflows are unlikely to happen. Nevertheless, a formal verification tool such as Frama-C/WP does not know about the size of ETCS telegrams and therefore needs this information.
- The assigns clause expresses that `Bitwalker_Peek` will not change any memory location outside its scope. This means in particular that `Bitwalker_Peek` will not have any side effects.

Behavior for invalid bit sequences The behavior `invalid_bit_sequence` describes the situation where the specified bit sequence does not fit into the underlying bit stream.

- The assumes clause describes the conditions to which this behavior applies. Note that we use the formulation

  ```
  (Start + Length) > 8 * BitstreamSize
  ```

  in order to describe an invalid bit sequence whereas the original implementation in Listing 3.1 used the expression

  ```
  ((Start + Length - 1) >> 3) >= BitstreamSize
  ```

  One difference is that we reformulate the division inherent in the shift operation as a multiplication. Also, switching to a strict inequality saves us the trouble to deal with a potential overflow in the term `(Start + Length - 1)` that occurs if both `Start` and `Length` are 0. Last but not least, the new expression is also shorter.
- The postcondition of this behavior is that `Bitwalker_Peek` is expected to return 0. Not surprisingly, we also request that no external memory locations are changed when this behavior is active.

Behavior for valid bit sequences The behavior `normal_case` describes the normal operation mode of `Bitwalker_Peek`.

- Note that the assumes clause is the negation of the assumes clause of the behavior `invalid_bit_sequence`.
- Again we specify that no assignments are to occur.
- At this point the formalization of the behavior of `Bitwalker_Peek` is incomplete. We only specify the rather weak postcondition that now overflow shall occur when computing the result. The complete formalization, based on Formula (2) on Page 25, will be part of a later release of this document.

Relationship of both behaviors The specification contains also statements about the relationship of the behaviors `normal_case` and `invalid_bit_sequence`.

- The clause **`complete behaviors`** expresses that the assumptions of both behaviors cover all admissible input values according to the general preconditions.
- The clause **`disjoint behaviors`** expresses that there are no input values that fit both behaviors.

These clauses, which support the writing complete and non-contradictory specifications, will be checked by Frama-C/WP.

3.5.2 Code annotations for `Bitwalker_Peek`

Listing 3.5 shows our modified version of `Bitwalker_Peek`. There are several reasons for these modifications:

- Loop invariants and static assertions had to be inserted into the source code to support the verification.
- Some shift operations were rewritten as divisions/multiplications to be more similar to the specification.
- The loop was rewritten so that loop index starts at 0.
- We felt that the shorter variable names make the source code more legible.

In order to ensure that the refactored code behaves as the original on we checked both with our test cases (see Section 3.4).

Of course, rewriting the implementation while verifying it may appear odd. Ideally, the verification tool should take the code as it is. However, as we have seen when discussing the specification, the expression to check whether the bit sequence is valid could be reformulated so that it does not raise unintended run time errors. Moreover, our refactoring removed an unnecessary cast (see Section 3.2.1).

Here are some additional notes on Listing 3.5.

- We added a (static) ACSL assertion that indicates whether Frama-C/WP is "aware" that `UINT64_MAX` equals $2^{64} - 1$.
- We added the following small helper function for converting a given "global" bit index into a "local" bit index that is used for right shifts.

```
/*@
    requires d > 0;

    assigns \nothing;

    ensures 0 <= \result < d;
*/
static inline
unsigned int inverse_modulo(unsigned int n, unsigned int d)
{
  return d - 1 - (n % d);
}
```

- There are several loop invariants and one loop variant. The latter is necessary for Frama-C/WP to decide whether the loop terminates.

  We mention here only the loop invariant that asserts that in the i-th iteration the value `retval` is less than 2^i. This, together with the precondition that `Length` is less than 64, is essential to ensure that no arithmetic overflow can occur when computing the return value of `Bitwalker_Peek`.

```
#include "Peek.h"

uint64_t Bitwalker_Peek(unsigned int Start,
                        unsigned int Length,
                        uint8_t Bitstream[],
                        unsigned int BitstreamSize)
{
  if ((Start + Length)  > 8 * BitstreamSize)
  {
    return 0;
  }

  //@ assert UINT64_MAX == (1 << 64) - 1;
  uint64_t retval = 0;

  /*@
    loop invariant 0 <= i <= Length;
    loop invariant 0 <= retval < (1 << i);
    loop assigns i, retval;
    loop variant Length - i;
  */
  for (unsigned int i = 0; i < Length; i++)
  {
    unsigned int pos = Start + i;
    unsigned int byte_index = pos / 8;
    unsigned int bit_index  = inverse_modulo(pos, 8);

    // treat as unsigned int for Frama-C
    unsigned int shifted = Bitstream[byte_index] >> bit_index;
    unsigned int bit_as_byte = shifted & 1;
    //@ assert bit_as_byte == 0 || bit_as_byte == 1;

    retval = 2 * retval + bit_as_byte;
  }

  return retval;
}
```

Listing 3.5. Implementation of `Bitwalker_Peek` with ACSL loop invariants

3.5.3 Formal specification of `Bitwalker_Poke`

Listing 3.6 shows an ACSL contract for the main operation modes of `Bitwalker_Poke`. Again we have labeled some properties of the contract and use for some variables shorter names than the original implementation.

```
#include "Bitwalker.h"

/*@
  requires writeable_bitstream:
           \valid(Bitstream + (0..BitstreamSize-1));
  requires valid_length: 0 <= Length < 64;
  requires no_overflow_1: Start + Length <= UINT_MAX;
  requires no_overflow_2: 8 * BitstreamSize <= UINT_MAX;

  assigns Bitstream[0..BitstreamSize - 1];

  behavior  invalid_bit_sequence:
    assumes (Start + Length)  > 8 * BitstreamSize;
    assigns \nothing;
    ensures \result == -1;

  behavior  value_too_big:
    assumes (1 << Length) <= Value &&
            (Start + Length) <= 8 * BitstreamSize;
    assigns \nothing;
    ensures \result == -2;

  behavior  normal_case:
    assumes Value < (1 << Length) &&
            (Start + Length) <= 8 * BitstreamSize;
    assigns Bitstream[0..BitstreamSize - 1];
    ensures \result == 0;

  complete behaviors;
  disjoint behaviors;
*/
int Bitwalker_Poke (unsigned int Start,
                    unsigned int Length,
                    uint8_t Bitstream[],
                    unsigned int BitstreamSize,
                    uint64_t Value);
```

Listing 3.6. Formal Specification of `Bitwalker_Poke`

The contract is structured as follows.

Default behavior The default of `Bitwalker_Poke` behavior is very similar to that of `Bitwalker_Peek`. The main difference is that `Bitwalker_Poke` writes into the array passed as argument.

- The property `writeable_bitstream` is formulated using the built-in ACSL predicate `\valid`. This expresses that all addresses starting at `Bitstream` and with offsets in the range `0..BitstreamSize-1` can be safely dereferenced for both *reading and writing*.
- The property `valid_length` expresses the requirement that only bit sequences with a length less than 64 are to be written.
- The two `overflow` properties request that no arithmetic overflow shall occur for the expressions `Start + Length` and `8 * BitstreamSize`.
- The assigns clause expresses that `Bitwalker_Poke` will write into a part of the array passed as argument. Apart from this assignment `Bitwalker_Poke` will not have any side effects.

Behavior for invalid bit sequences The behavior `invalid_bit_sequence` describes the situation when the specified bit sequence does not fit into the underlying bit stream.

The postcondition of this behavior is that `Bitwalker_Poke` is expected to return -1. We also strengthen the default assigns clause by requesting that no external memory locations are changed when this behavior applies.

Behavior for values that do not fit into the bit sequence The behavior `value_too_big` describes the case when the value to be converted into a bit sequence needs more bits than is provided by the (otherwise valid) bit sequence.

`Bitwalker_Poke` is then expected to return -2. No external memory locations are to be changed when this behavior is active.

Behavior for the normal case The behavior `normal_case` describes the normal operation mode of `Bitwalker_Poke`. This behavior assumes that the value to be converted is less than $2^{\texttt{Length}}$ and, of course, that only valid bit sequences are considered.

Since we concentrate on the absence of run time errors we only specify the range in the bit stream that is to be modified by `Bitwalker_Poke`. Note that the **`assigns`** clause describes the *bytes* that are allowed to be changed by `Bitwalker_Poke`, not the exact bits.

Relationship of the behaviors The contract of `Bitwalker_Poke` consists of the three named behaviors `normal_case`, `invalid_bit_sequence`, and `value_too_big`. These behaviors are *complete*, meaning that they cover all the input values of the default behavior. Another verification goal is to show that these three behaviors exclude each other.

3.5.4 Code annotations for `Bitwalker_Poke`

Listing 3.7 shows our modified version of `Bitwalker_Poke`.

```
#include "Poke.h"

int Bitwalker_Poke(unsigned int Start,
                   unsigned int Length,
                   uint8_t Bitstream[],
                   unsigned int BitstreamSize,
                   uint64_t Value)
{
  if ((Start + Length)  > 8 * BitstreamSize)
  {
    return -1;  // error: invalid_bit_sequence
  }

  // compute pow2(Length)
  const uint64_t MaxValue = (((uint64_t) 1) << Length);

  if (Value >= MaxValue)
  {
    return -2;  // error: value_too_big
  }

  /*@
    loop invariant 0 <= i <= Length;
    loop assigns i, Value, Bitstream[0..BitstreamSize-1];
    loop variant i;
  */
  for (unsigned int i = Length; i > 0; i--)
  {
    unsigned int pos = Start + i - 1;
    uint8_t mask = 1 <<  inverse_modulo(pos, 8);

    if ((Value % 2) == 0)
    {
      Bitstream[pos / 8] &= ~mask;
    }
    else
    {
      Bitstream[pos / 8] |=  mask;
    }

    Value /= 2;
  }

  // assert Value == 0;
  // We should prove this at one point because it would show
  // that we have consumed all bits of Value.

  return 0;
}
```

Listing 3.7. Implementation of `Bitwalker_Poke` with loop invariants

The reasons for modifications of `Bitwalker_Poke` are similar to those discussed in Section 3.5.2.

- Loop invariants had to be inserted into the source code to support the verification.
- Most shift operations were rewritten as divisions/multiplications to be more similar to the specification. In particular, we have omitted the helper array `BitwalkerBitMaskTable`. This has the advantage, at least from a verification point of view, that we do not have to deal with aliasing issues between this array and the array `Bitstream`.
- The loop was rewritten so that loop index starts at `Length` and that no casts to **`int`** are necessary.
- Again we used the shorter variable names already introduced in Section 3.5.2.
- Instead of testing that `Value` is greater than $2^{\texttt{Length}} - 1$ we use the briefer test that it is greater or equal than $2^{\texttt{Length}}$.

3.6 Results of formal verification with Frama-C/WP

In this section, we present the current state of the verification results for `Bitwalker_Peek` and `Bitwalker_Poke`. Here is a list of options with which we called the Frama-C/WP analysis in its version `Neon 20140301`.[4] For a detailed description of these options we refer to the documentation of Frama-C [10, 5].

```
-wp
-warn-signed-downcast
-warn-signed-overflow
-warn-unsigned-downcast
-warn-unsigned-overflow
-wp-rte
-wp-script 'wp0.script'
-wp-model Typed+ref
-wp-timeout 10
-wp-steps 2000
-wp-par 1
-wp-prover alt-ergo
-wp-prover cvc4
```

Table 3.1 lists how many of the generated proof obligations could be verified by theorem provers.

Function	proof obligations	verified obligations	verification rate	Qed	Alt-Ergo	CVC4	Coq
`Bitwalker_Peek`	74	72	97%	52	18	2	0
`Bitwalker_Poke`	87	85	97%	53	26	6	0

Table 3.1. Verification results for `Bitwalker_Peek` and `Bitwalker_Poke`

In the case of `Bitwalker_Peek`, two out of 72 proof obligations could not be verified automatically and will have to be handled by the interactive theorem prover Coq. The unproven obligations are related to the loop invariant that states that in iteration i the value `retval` is less than 2^i (see Listing 3.5). In the case of `Bitwalker_Poke`, the two unproven obligations are related to an issue in how Frama-C treats bitwise-and operations.

Table 3.1 also lists how many proof obligations were discharged by the various provers.

- Qed, which is a built-in simplifier of Frama-C/WP, discharges most of the obligations
- Alt-Ergo, which is the default theorem prover of Frama-C/WP, can deal with most of the remaining obligations
- CVC4, which is one of many external theorem provers that can be used with Frama-C/WP, discharges for `Bitwalker_Poke` two proof obligations that could be tackled neither by Qed nor Alt-Ergo
- Coq, which is an interactive theorem prover has not been used so far but will be employed to deal with the two remaining proof obligations of `Bitwalker_Peek`

[4] See `http://frama-c.com/download/frama-c-Neon-20140301.tar.gz`

3.7 Open issues

At this stage, only `Bitwalker_Peek` and `Bitwalker_Poke` have been formally specified with ACSL. Even for these two functions, a detailed formalization of bit operations is missing. This means, in particular, that we are not in the position to verify that for the normal operational modes `Bitwalker_Peek` and `Bitwalker_Poke` are *inverse* to each other. On the other hand, the specification covers already the main operational modes and provides a good foundation to show under which circumstances no run time errors will occur. Yet, there remain several unproven proof obligations for the `Bitwalker` functions. These obligations are also related to an insufficient formalization of bit operations.

It is important to keep in mind that the source code was modified to some extent. The main reasons for this were

- the need to rephrase some code constructs so that they are less susceptible to run time errors
- to simplify, from the point of view of Frama-C/WP, some bit operations
- to accommodate loop annotations and static assertions that are necessary for the formal verification

Fraunhofer FOKUS and CEA LIST will continue to work together to improve Frama-C/WP's capabilities to deal with bit operations. Here is a list of issues that Fraunhofer FOKUS came across while verifying the `Bitwalker`. The issues have been reported by Fraunhofer FOKUS in Frama-C's bug tracking systems.[5]

ID	Description
0001750	Frama-C/WP fails to discharge simple bit operation for small integer types
0001751	Frama-C/WP "forget" a proven assertion
0001761	Check that all occurrences of `*p` in assigns are guarded by a `\valid(p)` in requires
0001769	Unproven rte assertions for bit complement

In this context, we are are also investigating the use of the *interactive theorem prover* Coq to deal with unproven verification conditions. Using Coq's rich support for proof manipulation will very likely simplify the task of discharging the remaining proof obligations.

[5] See `https://bts.frama-c.com`

4 Static Analysis of Bitwalker

4.1 Introduction

In this chapter we describe our work on the static code analysis of the bitwalker code provided in [validation repository]

Our aim is to discover programing errors, obtain code metrics (lines of code, lines of code/lines of comments, cyclomatic complexity, Halsted metrics, class inherance tree and others) and verify the C11 standard and some subset of rules defined in the MISRA C Standard. That is, we focus on the different aspects of the source code to ensure the quality of the code in various perspectives.

The code metrics help understanding the complexity of the code and can lead to code changes. The complexity metrics allows us to identify particularly complex program areas that it would be desirable to redesign, and where problems that will appear in the maintenance phase are likely focused. For example, the cyclomatic complexity or the number of paths, is a software quality metric that quantifies the complexity of a program and also indicates the number of test cases that would have to be written to execute all paths in a program. However, the cyclomatic complexity only considers the decision structure of a program, not consider the complexity of nesting. There are more complexity metrics that takes into account the degree of nesting of a program or that consider the volumen and the program level like the Halsted metrics. The conjunction of the complexity metrics are an important indicator of the code readability, maintainability and portability, and the more complex the code is, more likely it will contain masked bugs.

CENELEC Standard identifies techniques and measures for 5 levels of software safety integrity and requires the use of a package of techniques and their correct application appropriate to the software safety integrity level.

Six different static analysis tools have been used during the code verification activities in order to assess the quality of the results, ensure code quality and cover different techniques and metrics high recommended by CENELEC Standard. The selected tools are:

- **Resource Standard Metrics (RSM)**: a source code metrics and quality analysis tool
- **LocMetrics**: a simple tool for counting lines of code in C#, Java, and C++
- **Understand**: a reverse engineering, documentation and metrics tool for C and C++ source code. It offers code navigation using a detailed cross reference, a syntax colorizing "smart" editor, and a variety of graphical reverse engineering views.
- **Clang Static Analyzer**: The Clang Static Analyzer consists of both a source code analysis framework and a standalone tool that finds bugs in C and Objective-C programs.
- **CPPcheck**: a static analysis tool for C, C++ code. Unlike C, C++ compilers and many other analysis tools it does not detect syntax errors in the code. Cppcheck primarily detects the types of bugs that the compilers normally do not detect.

- **Testwell CMT++**: Based on the static properties of the program code CMT++ gives estimates how error prone the program source code is due to its complexity, how long it will take to understand the code, what is the logical volume of the code, etc ...

Finally, according to the results obtained by using the tools, we will present some conclusions.

4.2 Resource Standard Metrics -RSM- Results

In this section we provide the results obtained with the [RSM] tool.

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

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

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

As mentioned previously CENELEC Standard requires the use of a package of techniques. With the use of the RSM tool the following Cenelec Standard techniques have been covered:

- Limited Size and Complexity in Functions, Subroutines and Methods (High Recommended)
- Coding Standard (Mandatory): At this point the fulfillment of some of the MISRA-C Standard rules has been checked.

4.2.1 Quality Metrics

As well as having intrinsic and user definied quality notices, RSM tool is mapped to the MISRA C Industry Standard. Taking into account the intrisic quality notice and the user defined quality notices the RSM tool covers 40.16% of [MISRA C] rules.

The following table shows the intrinsic Quality Notices for C language that RSM tool checks.

Table 4.1. 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.
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.

Table 4.1. Quality Notices

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 ensure 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.
Quality Notice No. 16 Emit a quality notice when function 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. 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.

Table 4.1. Quality Notices

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.
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.

Table 4.1. Quality Notices

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 code.	**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, some user defined quality notices are included in the rsm_udqn.cfg file. The table below shows those that are active and defined for C language.

Table 4.2. User Defined Quality Notices

User Defined Quality Notice No. 102 Emit a quality notice when dynamic memory using malloc is not initialized.	**User Defined Quality Notice No. 103** Emit a quality notice when the realloc function has been identified.
User Defined Quality Notice No. 104 Emit a quality notice when a line containing just a semicolon has been identified.	**User Defined Quality Notice No. 105** Emit a quality notice when a symbolic constant using #define has been identified
User Defined Quality Notice No. 107 Emit a quality notice when a double ;; has been identified.	**User Defined Quality Notice No. 109** Emit a quality notice when a double pointer indirection has been identified
User Defined Quality Notice No. 116 Emit a quality notice if Pointer variable uninitialized.	**User Defined Quality Notice No. 125** Emit a quality notice when a data member in the header file is not of the form m_*

Taking into account the quality notices mentioned above, a table that indicates the total quality profile (Summary by notice type) for the bitwalker code is shown. This result is especially useful for determining the overall internal code quality.

Table 4.3. Quality Profile

Type	Count	Percent	Quality Notice
1	38	9.57	Physical line length > 80 characters

Table 4.3. Quality Profile

Type	Count	Percent	Quality Notice
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_*

More detailed information regarding to in what line, function or file the quality notices have been detected is provided in the [bitwalker_functional_quality_metrics file].

4.2.2 Complexity Metrics

Reflecting on elements that can contribute to increase the complexity of a program and influencing in its maintenance, four elements are identified:

- Program Size
- Data Structure
- Data Flow
- Control Flow

4.2.2.1 Program Size Metrics

Very large programs are complex even if only be for the large amount of information to be considered in order to understand them. So a first measure of the code complexity is given by its size. This size can be determined using the following metrics:

- Number of lines
- Halsted metrics (See 4.7)

Count the number of code lines in a program is a simple way to measure its size. The main problem with this metric is to decide what we consider as line. The reason is that there is no standard definition of what a line of code is. Do comments count? Are data declarations included? What happens if a statement extends over several lines? – These are the questions that often arise. According to the criteria that we follow a different metric will be obtained.

For example, in C language, a line of code can be:

- an statement, instruction finished in a jump line

- an statement, instruction terminated with a semi-colon
- any line of the program terminated with a new line (comments included)

As there is no standard definition and the definitions of these metrics are tied to specific computer languages, a definition of how the RSM tool considers these code metrics is indicated below.

- An effective line of code is the measurement of all lines that are not comments, blanks or standalone braces or parenthesis. RSM counts the instances of lines that contain a single brace and parenthesis and creates a metric for effective lines of source code, eLOC. This metric is the result of subtracting the single braces and parenthesis from the LOC measurement.
- Logical lines of code represent a metrics for those line of code which form code statements. These statements are terminated with a semi-colon. The control line for the "for" loop contain two semi-colons but accounts for only one semi colon.
- Comments: RSM counts a comment line as any physical line that contains a comment.

Taking into account these criterias the following size metrics are obtained:

Table 4.4. File Summary

Metrics	Bitwalker.h	Bitwalker.c	opnETCS.h	opnETCS _Decoder.h
LOC[6].	15	58	884	62
eLOC[7]	15	40	823	62
lLOC[8]	11	28	760	61
Comment	16	29	822	15
Lines	41	109	1249	84

The following table describes some recommendations for the lines-of-code measures:

Table 4.5. Recommendations

Measures	Values	Comments
Function length	4-40 program lines	A function definition contains at least a prototype, one line of code, and a pair of braces, which makes 4 lines. A function longer than 40 program lines probably implements many functions. Functions containing one selection statement with many branches are an exception to this rule. Decomposing them into smaler functions often decreases readability.

[6]Lines of Code

[7]Effective Lines of Codes

[8]Logical Statements Lines of Code: represent a metrics for those line of code which form code statements. These statements are terminated with a semi-colon. The control line for the "for" loop contain two semi-colons but accounts for only one semi colon

Table 4.5. Recommendations

Measures	Values	Comments
File length	4-400 program lines	The smallest entity that may reasonably occupy a whole source file is a function, and the minimum length of a function is 4 lines. Files longer than 400 program lines (10..40 functions) are usually too long to be understood as a whole.
Comments Percentage	30%-75%	If less than one third of a file is comments the file is either very trivial or poorly explained. If more than 75% of a file are comments, the file is not a program but a document. In a well-documented header file percentage of comments may sometimes exceed 75%

By analyzing the results, one can observe the Bitwalker.c file fulfills the recommendations in relation to the file length. Although the comments percentage (26%) is a little bite under the recommended value, this do not indicate a poor documentation of source code.

4.2.2.2 Control Flow Metrics

The possibility that the execution flow of a program follows different paths depending on whether or not certain conditions are met, increases the difficulty to understand what the program do in each of the situations that may occur.

One metric that addresses the complexity of the control flow is the **Cyclomatic complexity**.

The cyclomatic complexity metric measures the complexity of the code by counting the number of independent paths through a piece of code-by counting the number of decision points. The decision point is where a choice can be made during execution; this gives rise to different paths through the code. Decision points arise through if statements and through while, do while and for loops. A single switch or try statement can also add many more decision points. This metric can either be determined by counting the regions, nodes and edges or number of predicate nodes (branching points) with a flow graph.

The following equations defined McCabe Cyclomatic Complexity:

- The number of regions in a flow graph.
- V(g) = E - N + 2P, where E are the edges, N are the nodes and P nodes without outgoing path.
- V(g) = P + n, where P are the predicate nodes and n the number of output.

When the graph is strongly connected, a simplified formula to calcule the cyclomatic complexity is use: V(g) = P + 1, where P are the predicate nodes.

The result obtained in the calculation of the cyclomatic complexity also determines the upper bound on the number of tests that must be performed to ensure that each statement is executed at least once.

At following the results of some complexity metrics obtained by the RSM tool are shown:

Table 4.6. Functional Summary

Metrics	Bitwalker.c
File Function Count	7
Total Function LOC	49
Total Function eLOC	31
Total Function lLOC	27
Total Function Params	20
Total Cyclo Complexity	13
Total Function Pts LOC	0.5
Total Function Pts eLOC	0.3
Total Function Pts lLOC	0.2
Total Function Return	10
Total Function Complex	43
Max Function LOC	16
Max Function eLOC	12
Max Function lLOC	9
Average Function LOC	7.00
Average Function eLOC	4.43
Average Function lLOC	3.86
Max Function Parameters	5
Max Function Returns	3
Max Interface Complex	8
Max Cyclomatic Complex	5
Max Total Complexity	13
Avg Function Parameters	2.86
Avg Function Returns	1.43
Avg Interface Complex	4.29
Avg Cyclomatic Complex	1.86
Avg Total Complexity	6.14

The interface complexity is defined by RSM as the number of input parameters to a function plus the number of return states from that function. Class interface complexity is the sum of all function interface complexity metrics within that class.

The Maximun total complexity is the addition of Maximun Interface and Cyclomatic complexities and the total Cyclomatic complexity is calculated as the sumn of the cyclomatic complexity of each function of the file.

Knowing that a program has a high value of cyclomatic complexity (total Cyclomatic complexity) does not provide us enough info to decide what actions to take to improve our software. This

occurs due to there is not an approximate threshold reference value for total cyclomatic complexity since not all software has the same size. However we can say that the cyclomatic complexity of each function should not exceed a certain value.

Due to this, a more detailed Complexity analysis per function is provided at following.

Table 4.7. Function Metrics

Bitwalker_Peek			
Cyclomatic Complexity Vg Detail:			
Function Base			1
Loops for / foreach			1
Conditional if / else if			1
Param: 4	Return: 2	Cyclo Vg: 3	Comment: 5
LOC: 12	eLOC: 8	lLOC: 7	Lines: 19
Bitwalker_Poke			
Cyclomatic Complexity Vg Detail:			
Function Base			1
Loops for / foreach			1
Conditional if / else if			3
Param: 5	Return: 3	Cyclo Vg: 5	Comment: 6
LOC: 16	eLOC: 12	lLOC: 9	Lines: 23
Bitwalker_IncrementalWalker_Init			
Param: 4	Return: 1	Cyclo Vg: 1	Comment: 0
LOC: 5	eLOC: 3	lLOC: 3	Lines: 5
Bitwalker_IncrementalWalker_Peek_Next			
Param: 2	Return: 1	Cyclo Vg: 1	Comment: 1
LOC: 5	eLOC: 3	lLOC: 3	Lines: 6
Bitwalker_IncrementalWalker_Peek_Finish			
Param: 1	Return: 1	Cyclo Vg: 1	Comment: 0
LOC: 3	eLOC: 1	lLOC: 1	Lines: 3
Bitwalker_IncrementalWalker_Poke_Next			
Param: 3	Return: 1	Cyclo Vg: 1	Comment: 1
LOC: 5	eLOC: 3	lLOC: 3	Lines: 6
Bitwalker_IncrementalWalker_Poke_Finish			
Param: 1	Return: 1	Cyclo Vg: 1	Comment: 0
LOC: 3	eLOC: 1	lLOC: 1	Lines: 3

After calculating the cyclomatic complexity the risk involved can be determined using the following table:

Table 4.8. Mc Cabe cyclomatic Complexity Reference table

Cyclomatic Complexity	Risk Evaluation
1-10	Low risk
11-20	More complex, Moderate risk
21-50	Complex, High Risk
>50	Not testable, Very High Risk

If we cross the values obtained in the analysis with the indicative table we can see that all functions are under 10, so we speak of simple functions with little logic and with low risk.

In addition to the Limited Size and Complexity in Functions, Subrutines and Methods and Coding Standard techniques, at following we can see that taking into account the modular approach where one of its rule mentions that it shall specify a restriction for the number of paramenters (normally 5) the Parameter Number Limit is fulfilled.

Furthermore, from the previous definition of recommended values for lines of code measures (see 4.5), we can see there is not documentation for some functions.

Now, an example of the cyclomatic complexity calculation for the bitwalker_Poke function is shown to compare the correctness of these results .

```
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 4.1. Bitwalker_Poke

The control flow generated from the bitwalker_Poke function would look like figure 4.1.

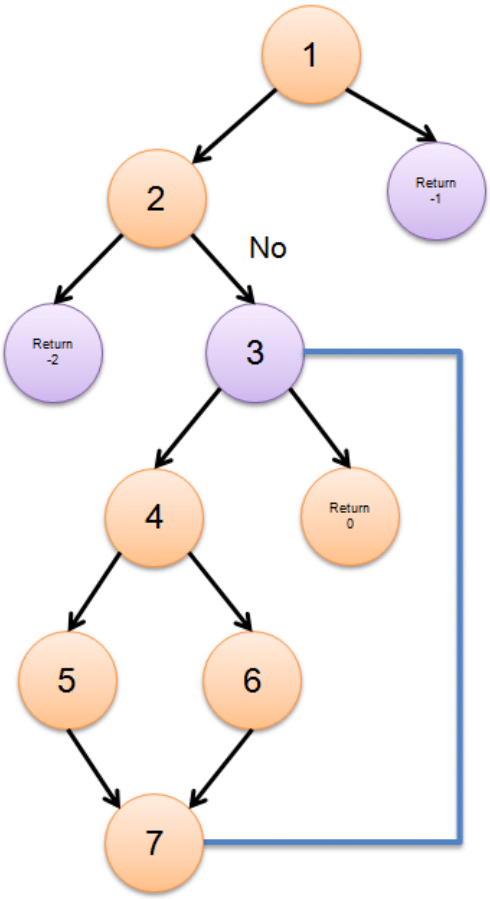


Figure 4.1. Bitwalker_Poke Flow

In this flow, 4 predicated nodes are displayed so, taking into account the equation V(g) = P + 1, where P are the predicate nodes, we see that the cyclomatic complexity of this function is V(g)=5.

4.3 LocMetrics tool Results

[LocMetrics] tool 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 4.9. 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

Table 4.9. LocMetrics Tool Results

File	LOC	SLOC-P	SLOC-L	MVG	BLOC	C&SLOC	CLOC	CWORD	HCLOC	HCWORD
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

4.4 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. Understand also provides a way to check the code using coding Standard to avoid potencial errors. With this tool SQS has checked MISRA-C:2004 and code metrics (lines of code, complexity, object cross reference, invocation tree, Unused Items and others). The high recomended and mandatory techniques identified by CENELEC Standard covered by the tool are:

- Coding Standard (Mandatory)
- Limited Size and Complexity in Functions, Subroutines and Methods (High Recomended)
- Data Flow Analysis technique (High Recomended)
- Control Flow Analysis technique (High Recomended)

The detailed static analysis report is available in the [VnVUserStories folder]

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**
  - 5.1 (req): Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
  - 5.2 (req): 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): `typedef`s 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 shall be defined at block scope if they are only accessed from within a single function.
  - 8.8 (req): An external object or function shall be declared in one and only one file.
  - 8.9 (req): An identifier with external linkage shall have exactly one external definition.
  - 8.10 (req): 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 a `<filename>` sequence.
  - 19.4 (req-): C macros shall only expand to a braced initializer, 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 inloffsetof, 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.

After a review of the subset of MISRA-C rules taking into account project requirements and sector standard or best practices it is necessary to decide which of some of them are not to be implemented/approved due to its application can get worse understanbility of the code and which other rules of other standard will be applied.

The table below shows the non approved MISRA-C rules.

Table 4.10. Status of MISRA Rules

MISRA Rule	Status
Global 5.1	no recommended
Global 5.6	no recommended

The results of the MISRA Rules are the following:

```
Begin Analysis: jueves, 21 de noviembre de 2013 13:28:18
Begin Global Check Phase
Global: 5.1 Identifiers shall not rely on the significance of more than 31 characters: Violations found
Global: 5.4 A tag name shall be unique.: Violations found
Global: 5.6 No identifier in one name space should have the same spelling as an identifier in another
name space.: Violations found
Global: 5.7 No identifier name should be reused: Violations found
Global: 8.10 prefer internal linkage over external whenever possible: Violations found
Global: 8.11 use static keyword for internal linkage: Violations found
Global: 8.9 identifier with external linkage shall have exactly one external definition.: Violations found
End Global Check Phase
Begin File Check Phase
File: Bitwalker.h: Violations found
File: opnETCS.h: Violations found
File: main.c: Violations found
File: Bitwalker.c: Violations found
End File Check Phase
Begin Clang Check Phase
End Clang Check Phase
End Analysis: jueves, 21 de noviembre de 2013 13:28:34
Analysis Summary:
Files: 5
Checks: 55
Violations Found: 1965
Violations Ignored: 0

Violations Remaining: 1965
```

Figure 4.2. MISRA-C Rules results

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

Table 4.11. Summary of detected 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

A detailed information about the file, entity, line, check, etc of all violations detected above can be found in the index files of [Results] and [Results2] folders.

In addition to the MISRA-C compliance checking, we also run code metrics analysis in order to ensure the correctness of the obtained results through the results comparation.

Below tables shows some different metrics per file and function. In order to understand the tables and to be able to compare the results obtained with the different tools the definition of the specific metrics is provided before the presentation of the corresponding table.

- Cyclomatic: The measure of the complexity of a function's decision structure. The cyclomatic complexity is also the number of basis, or independent, paths through a module.
- Modified Cyclomatic: cyclomatic except each case statement is not counted;the entire switch counts as 1.
- Strict: Cyclomatic complexity except each short-circuit operator adds 1 to the complexity.
- Essential Complexity: cyclomatic complexity after structured programming constructs have been removed.
- Nesting: maximum nesting level of control constucts (if, while,etc.)
- Count Path: Number of unique paths through a body of code (not counting gotos or abnormal exits

Table 4.12. Function Complexity metrics

Bitwalker_Peek	
Cyclomatic:	3
Modified Cyclomatic:	3
Strict Cyclomatic:	3
Essential:	1
Max Nesting:	1
Count Path:	3
Bitwalker_Poke	

Table 4.12. Function Complexity metrics

Cyclomatic:	5
Modified Cyclomatic:	5
Strict Cyclomatic:	5
Essential:	3
Max Nesting:	2
Count Path:	5
Bitwalker_IncrementalWalker_Init	
Cyclomatic:	1
Modified Cyclomatic:	1
Strict Cyclomatic:	1
Essential:	1
Max Nesting:	0
Count Path:	1
Bitwalker_IncrementalWalker_Peek_Next	
Cyclomatic:	1
Modified Cyclomatic:	1
Strict Cyclomatic:	1
Essential:	1
Max Nesting:	0
Count Path:	1
Bitwalker_IncrementalWalker_Peek_Finish	
Cyclomatic:	1
Modified Cyclomatic:	1
Strict Cyclomatic:	1
Essential:	1
Max Nesting:	0
Count Path:	1
Bitwalker_IncrementalWalker_Poke_Next	
Cyclomatic:	1
Modified Cyclomatic:	1
Strict Cyclomatic:	1
Essential:	1
Max Nesting:	0
Count Path:	1
Bitwalker_IncrementalWalker_Poke_Finish	
Cyclomatic:	1
Modified Cyclomatic:	1

Table 4.12. Function Complexity metrics

Strict Cyclomatic:	1
Essential:	1
Max Nesting:	0
Count Path:	1

Here are some remarks about how the Understand tool defines and take into account the following code metrics:

- Lines: total lines (in a function or file or project)
- Comment Lines: total lines that have comments on them
- Blank Lines: total lines without any code/comment
- Code Lines: total lines that have any code on them
- Executable Lines: total lines that have executable code on them
- Declarative Lines: total lines that have declarative code on them
- Execution Statements: total statements in executable code
- Declaration Statements: total statements in declarative code
- Ratio Comment/Code: comment lines / code lines

Table 4.13. File Metrics

Metrics	**Bitwalker.h**	**Bitwalker.c**	**opnETCS.h**	**opnETCS _Decoder.h**
Lines:	41	109	1249	84
Comment Lines:	20	33	822	20
Blank Lines:	7	23	180	2
Preprocessor Lines:	4	1	1	1
Code Lines:	11	57	883	61
Inactive Lines:	0	0	0	0
Executable Code Lines:	0	30	0	0
Declarative Code Lines:	11	15	822	61
Execution Statements:	0	28	0	0
Declaration Statements:	11	15	760	61
Ratio Comment/Code:	1.82	0.58	0.93	0.33
Units	0	7	0	0

Table 4.14. Function code Metrics

Bitwalker_IncrementalWalker_Init	
Lines:	6
Comment Lines:	0
Blank Lines:	0
Code Lines:	6
Inactive Lines:	0
Executable Code Lines:	3
Declarative Code Lines:	1
Execution Statements:	3
Declaration Statements:	0
Ratio Comment/Code:	0.00
Bitwalker_IncrementalWalker_Peek_Finish	
Lines:	4
Comment Lines:	0
Blank Lines:	0
Code Lines:	4
Inactive Lines:	0
Executable Code Lines:	1
Declarative Code Lines:	1
Execution Statements:	1
Declaration Statements:	0
Ratio Comment/Code:	0.00
Bitwalker_IncrementalWalker_Peek_Next	
Lines:	7
Comment Lines:	1
Blank Lines:	0
Code Lines:	6
Inactive Lines:	0
Executable Code Lines:	3
Declarative Code Lines:	2
Execution Statements:	2
Declaration Statements:	1
Ratio Comment/Code:	0.17
Bitwalker_IncrementalWalker_Poke_Finish	
Lines:	4
Comment Lines:	0
Blank Lines:	0

Table 4.14. Function code Metrics

Code Lines:	4
Inactive Lines:	0
Executable Code Lines:	1
Declarative Code Lines:	1
Execution Statements:	1
Declaration Statements:	0
Ratio Comment/Code:	0.00
Bitwalker_IncrementalWalker_Poke_Next	
Lines:	7
Comment Lines:	1
Blank Lines:	0
Code Lines:	6
Inactive Lines:	0
Executable Code Lines:	3
Declarative Code Lines:	2
Execution Statements:	2
Declaration Statements:	1
Ratio Comment/Code:	0.17
Bitwalker_Peek	
Lines:	20
Comment Lines:	5
Blank Lines:	4
Code Lines:	13
Inactive Lines:	0
Executable Code Lines:	7
Declarative Code Lines:	4
Execution Statements:	7
Declaration Statements:	3
Ratio Comment/Code:	0.38
Bitwalker_Poke	
Lines:	24
Comment Lines:	6
Blank Lines:	4
Code Lines:	17
Inactive Lines:	0
Executable Code Lines:	11
Declarative Code Lines:	3

Table 4.14. Function code Metrics

Execution Statements:	12
Declaration Statements:	2
Ratio Comment/Code:	0.35

Taking into account control flow and data flow techniques some Uninitialized Items (items such as variables that are not initialized in the code), Unused Variables and Parameters items (items that are declared (and perhaps initialized) but never referenced other than that) and Unused Program Units have been identified. The Unused Program Units Report identifies program units that are declared but never used. However note that this listing in this report doesnt mean the system doesnt need this program unit.

Table 4.15. Unused Variables and Parameters

File	Item	Type of Item	Location
Bitwalker.c	Bitwalker_IncrementalWalker_Peek_Finish	Function	line 91
Bitwalker.c	Bitwalker_IncrementalWalker_Peek_Next	Function	line 82
Bitwalker.c	Bitwalker_IncrementalWalker_Poke_Finish	Function	line 106

Table 4.16. Uninitialized Items

File	Item	Location
Bitwalker.c	i	line 35
Bitwalker.c	i	line 60

Table 4.17. Unused Program Units

File	Item	Location
Bitwalker.c	Bitwalker_IncrementalWalker_Peek_Finish	line 91
Bitwalker.c	Bitwalker_IncrementalWalker_Peek_Next	line 82
Bitwalker.c	Bitwalker_IncrementalWalker_Poke_Finish	line 106

4.5 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 4.18. Aspects checked

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).

Table 4.18. Aspects checked

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.
deadcode.DeadStores	Check for values stored to variables that are never read afterwards.
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.

Taking into account the features checked by the tool, the following Cenelec Standard techniques have been covered:

- Boundary Value Analysis (High Recommended)

- Data Flow Analysis (High Recommended)

After run this analysis no violation has 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 4.3. Clang Analysis results

4.6 CPPcheck tool Results

Bitwalker folder has been analyzed statically by [CPPcheck] tool (Complying with the standard C11).

Cppcheck supports the following languages: C89, C99, C11 and a wide variety of static checks. The following features are provided:

- Out of bounds checking
- Check the code for each class
- Checking exception safety
- Memory leaks checking
- Warn if obsolete functions are used
- Check for invalid usage of STL
- Check for uninitialized variables and unused functions
- Check input/output operations
- Null pointer dereferencing

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.

With the use of this tool the following techniques recommended by CENELEC Standard are covered:

- Coding Standard (mandatory) (checked C11 standard)
- Boundary Value Analysis (High Recommended)
- Data Flow Analysis (High Recommended)

The results of the tool show that there are some verbose errors in the main file and some errors in the bitwalker.c file.

- repetitive verbose error regarding to Testwort variable is reassigned value before the old one has been used (lines 119, 120 and 121 in main.c)
- one error about the Testwort variable is assigned a value that is never used (line 122 in main.c).
- the funtions Bitwalker_IncrementalWalker_Peek_Finish (line 91 in bitwalker.c), Bitwalker_IncrementalWalker_Pee (line 82 in bitwalker.c) and Bitwalker_IncrementalWalker_Poke_Finish (line 106 in bitwalker.c) are never used,

The below figure shows the results commented previously:

Figure 4.4. cppcheck results

4.7 Testwell CMT++ Results

CMT++, Complexity Measures Tool for C/C++, is an easy-to-use code metrics tool for C and C++ languages. Also assembly code, either inlined in a C/C++ source file or separate assembly file, can be measured.

Based on the static properties of the program code CMT++ gives estimates how error prone the program sorce code is due to its complexity, how long it will take to understand the code, what the logical volume of the code is, how much code you have: physical lines, comment lines, program lines, statements, etc

CMT++ helps to estimate the overall maintainability of the code base and easily locate the complex parts of it.

In this case CMT++ is used to calculate:

- Basic code complexity metrics
  - McCabe's cyclomatic number
  - Halstead's metrics
  - Lines of code metrics
  - Some other metrics like: number of semicolons, number of function parameter, depth of control structure nesting
- Maintainability Index

4.7.1 Complexity Metrics

4.7.1.1 Program Size Metrics

As it was mentioned in 4.2.2.1 the number of lines and the Halsted metrics can be used to determine the program size.

Number of lines

The lines of code measures are the most traditional measures used to quantify software complexity. They are simple, easy to count, and very easy to understand. However, they do not take into account the intelligence content and the layout of the code.

CMT++ calculates the following lines-of-code metrics:

- LOCphy: number of physical lines
- LOCbl: number of blank lines (a blank line inside a comment block is considered to be a comment line)
- LOCpro: number of program lines (declarations, definitions, directives, and code)
- LOCcom: number of comment lines

Following the analysis conducted within the tool, the tables below summarizes the results:

Table 4.19. Lines of Code Metrics per file

File	LOCphy	LOCpro	LOCcom	LOCbl
Bitwalker.c	109	58	33	23

Table 4.20. Lines of Code Metrics per functions

Function	LOCphy	LOCpro	LOCcom	LOCbl
Bitwalker_Peek	20	13	5	4
Bitwalker_Poke	24	17	6	4
Bitwalker_IncrementalWalker_Init	6	6	0	0
Bitwalker_IncrementalWalker_Peek_Next	7	6	1	0
Bitwalker_IncrementalWalker_Peek_Finish	4	4	0	0
Bitwalker_IncrementalWalker_Poke_Next	7	6	1	0
Bitwalker_IncrementalWalker_Poke_Finish	4	4	0	0

Halsted metrics

Halstead complexity metrics were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module to measure a program module's complexity directly from source code.

Halstead's metrics are based on interpreting the source code as a sequence of tokens and classifying each token to be an operator or an operand. There are based on four basic measures:

- number of unique (distinct) operators (n_1)
- number of unique (distinct) operands (n_2)
- total number of operators (N_1)
- total number of operands (N_2).

Taking into account these measures the following metrics will be obtained:

- Program Vocabulary: $n = n_1 + n_2$
- Program Length: $N = N_1 + N_2$
- Program Difficulty: $D = (n_1/2) * (N_2/n_2)$
- Program Volume: $V = N * log_2(n)$
- Program Length: L = 1/V
- Effort to implement: E = V * D
- Time to implement: T = E / 18
- Number of delivered bugs: $B = (E^{2/3})/3000$

So Halsted metrics provide several metrics that focus on different aspects of software complexity. Furthermore, they allow the estimation of development and testing times (with parameter L*V), and difficulty of understanding (with parameter E).

Software complexity is usually analyzed with the indicators L, V and E due to:

- the volume is intended being a more accurate measure of the difficulty of understanding a program, taking into account not only its length but also the vocabulary. Halstead's volume (V) describes the size of the implementation of an algorithm.
- the program level gives an idea of the level of detail that it has been encoded
- effort can be used as a measure of program clarity since the effort required to produce a piece of software is primarily related to the difficulty to understand and implement it.

In the tables below are presented the results obtained per file and per functions:

Table 4.21. Halsted metrics 1 per file

File	L	n	n_1	n_2	N	N_1	N_2
Bitwalker.c	0.014	72	31	41	378	185	193

Table 4.22. Halsted metrics 2 per file

File	B	E	T	D	V
Bitwalker.c	1.024	170167.578	02:37:33	72.963	2332.232

Table 4.23. Halsted metrics 1 per functions

Function	L	n	n_1	n_2	N	N_1	N_2
Bitwalker_Peek	0.041	35	18	17	89	43	46
Bitwalker_Poke	0.028	42	23	19	120	62	58
Bitwalker_IncrementalWalker_Init	0.143	20	8	12	37	16	21
Bitwalker_IncrementalWalker_Peek_Next	0.116	20	9	11	39	18	21
Bitwalker_IncrementalWalker_Peek_Finish	0.278	11	6	5	12	6	6
Bitwalker_IncrementalWalker_Poke_Next	0.111	21	9	12	44	20	24
Bitwalker_IncrementalWalker_Poke_Finish	0.278	11	6	5	12	6	6

Table 4.24. Halsted metrics 2 per functions

Function	B	E	T	D	V
Bitwalker_Peek	0.166	11117.269	00:10:17	24.353	456.506
Bitwalker_Poke	0.267	22715.846	00:21:01	35.105	647.078
Bitwalker_IncrementalWalker_Init	0.036	1119.379	00:01:02	7.000	159.911
Bitwalker_IncrementalWalker_Peek_Next	0.043	1448.042	00:01:20	8.591	168.555
Bitwalker_IncrementalWalker_Peek_Finish	0.009	149.447	00:00:08	3.600	41.513
Bitwalker_IncrementalWalker_Poke_Next	0.048	1739.358	00:01:36	9.000	193.262
Bitwalker_IncrementalWalker_Poke_Finish	0.009	149.447	00:00:08	3.600	41.513

The volume of a function should be at least 20 and at most 1000. The volume of a parameter less one-line function that is not empty; is about 20. A volume greater than 1000 tells that the function probably does too many things.

The volume of a file should be at least 100 and at most 8000. These limits are based on volumes measured for files whose LOCpro and v(G) are near their recommended limits. The limits of volume can be used for double-checking.

Halstead's delivered bugs (B) is an estimate for the number of errors in the implementation. Delivered bugs in a file should be less than 2. Experiences have shown that, when programming

with C or C++, a source file almost always contains more errors than B suggests. The number of defects tends to grow more rapidly than B.

By analyzing the results, one can observe that all the Halsted metrics obtained in relation to functions and file are inside the recommendations.

4.7.1.2 Control Flow Metrics

Table 4.25. McCabe Cyclomatic Complexity

Function	ECC
Bitwalker_Peek	3
Bitwalker_Poke	5
Bitwalker_IncrementalWalker_Init	1
Bitwalker_IncrementalWalker_Peek_Next	1
Bitwalker_IncrementalWalker_Peek_Finish	1
Bitwalker_IncrementalWalker_Poke_Next	1
Bitwalker_IncrementalWalker_Poke_Finish	1

As a first conclusion, taking into account the reference table shown in Section 4.2.2.2 the values from the above table indicate a low risk functions and the matching with the results obtained with the previous tools.

4.7.2 Maintainability Index

Maintainability Index (MI) is a single-number value for estimating the relative maintainability of the code.

Maintainability Index is calculated with certain formulae from lines-of-code measures, McCabe measure and Halstead measures.

Actually there are three measures:

- MIwoc: Maintainability Index without comments
- MIcw: Maintainability Index comment weight
- MI: Maintainability Index = MIwoc + MIcw

The general formulae for MI is the following:

MIwoc = 171 - 5.2 * ln(aveV) -0.23 * aveG -16.2 * ln(aveLOC)

MIcw = 50 * sin(sqrt(2.4 * perCM))

MI = MIwoc + MIcw

where:

- aveV = average Halstead Volume per Module
- aveG = average extended cyclomatic complexity per Module
- aveLOC = average count of lines per Module
- perCM = average percent of lines of comments per Module

Table 4.26. Maintainability Index

Function	MIwoc	MIcw	MI
Bitwalker_Peek	90	35	125
Bitwalker_Poke	85	35	120
Bitwalker_IncrementalWalker_Init	115	0	115
Bitwalker_IncrementalWalker_Peek_Next	113	28	140
Bitwalker_IncrementalWalker_Peek_Finish	129	0	129
Bitwalker_IncrementalWalker_Poke_Next	112	28	140
Bitwalker_IncrementalWalker_Poke_Finish	129	0	129

After calculating the Maintainability Index the maintainability involved can be determined using the following reference table:

Table 4.27. Maintainability Index Reference table

Maintainability Index	Maintainability Evaluation
85 and more	good maintainability
65-85	moderate maintainability
< 65	difficult to maintain with really bad pieces of code (big, uncommented, unstructured) the MI value can be even negative

As a first conclusion, the values from the tables above indicate the functions and file have a good maintanability.

4.8 MISRA and Mü8004 Rules Comparation

This section analyses the requirements designed in MISRA-C : 2004 standard and Mü8004 standard and makes a comparison between rules they might have in common and describes the most important features of the ones they don't have in common.

ENVIRONMENT:

MISRA –C Environment Rule 1.1

MISRA-C Rule 1.1 All code shall conform to ISO/IEC 9899:1990 “Programming languages — C”, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.

This rule is not included in Mü8004 standard. It never mentions which version of C is applied.

MISRA –C Environment Rule 1.2

MISRA-C Rule 1.2 No reliance shall be placed on undefined or unspecified behaviour.

This rule is not included in Mü8004 standard. No requirement on behaviour is specified.

MISRA –C Environment Rule 1.3

MISRA-C Rule 1.3 Description: If a module is to be implemented in a language other than C, or compiled on a different C compiler, then it is essential to ensure that the module will integrate correctly with other modules.

This rule is not included in Mü8004 standard. Mü8004 standard establishes in rule 0.2.22 that it is not allowed to change the programming language inside a program module, but not between different modules.

MISRA –C Environment Rule 1.4

MISRA-C Rule 1.4 Description: The compiler/linker shall be checked to ensure that 31 character significance and case sensitive are supported for external identifiers. If the compiler/linker is not capable of meeting this limit, then use the limit of the compiler.

This rule is not included in Mü8004 standard, although in 0.2.9 rule it is referred that names and identifiers must be chosen in a way that differs significantly in the first 31 positions. This rule should be added.

MISRA –C Environment Rule 1.5

MISRA-C Rule 1.5 Description: Floating-point implementations should comply with a defined floating-point standard.

This rule is not included in Mü8004 standard and it would be useful adding it to overcome a wide range of problems associated with the use of floating-point arithmetics.

LANGUAGE EXTENSION:

MISRA –C Language extensions 2.1

MISRA-C Rule 2.1 Description: Where assembly language instructions are required it is recommended that they be encapsulated and isolated in either (a) assembler functions, (b) functions or (c) macros.

This rule is included in Mü8004 Rule 0.2.22, as it establishes that assembler implemented subroutines can be called from C and vice versa. It would be useful to point out that assembler language should not be embedded in normal code.

MISRA –C Language extensions 2.2

MISRA-C Rule 2.2 Description: Source code shall only use /*. . . */ style comments.

This is less restrictive in Mü8004 standard, as // comments are allowed in Mü8004 Rule 0.2.8 if it is supported by compiler. This is correctly focused as this way consistency is not in danger.

MISRA –C Language extensions 2.3

MISRA-C Rule 2.2 Description: The character sequence /* shall not be used within a comment.

This rule is included in Mü8004 Rule 0.2.8, as it establishes that comments must not be nested. This is an important requirement whom omission would cause critical errors otherwise.

MISRA –C Language extensions 2.4

MISRA-C Rule 2.4 Description: Sections of code should not be “commented out”

This rule is not included in Mü8004 standard. It would be useful to use conditional compilation (#if or #ifdef) for sections of source code not to be compiled, as start and end comment markers for this purpose is dangerous because C does not support nested comments.

CHARACTER SETS:

MISRA –C Character sets 4.1

MISRA-C Rule 4.1 Only those escape sequences that are defined in the ISO C standard shall be used.

Mü8004 standard does not include this rule. This rules is useful for code portabiliy.

MISRA –C Character sets 4.2

MISRA-C Rule 4.2 Trigraphs shall not be used.

Mü8004 standard does not include this rule. This rules is useful for code understanding. This rule should be mandatory.

IDENTIFIERS:

MISRA –C Identifiers 5.1

MISRA-C Rule 5.1 Description: Identifiers (internal and external) shall not rely on the significance of more than 31 characters.

This rule is included in Mü8004 Rule 0.2.9, as it establishes that names and identifiers must be chosen in a way that they differ significantly in the first 31 positions.

MISRA –C Identifiers 5.2

MISRA-C Rule 5.2 Description: Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

This rule is not included in Mü8004 standard, but it would be useful adding it to avoid confusion between identifiers in the code.

MISRA –C Identifiers 5.3

MISRA-C Rule 5.3 Description: A typedef name shall be a unique identifier.

This rule is not included in Mü8004 standard, but it would be useful adding it to avoid the reuse this names within a program.

MISRA –C Identifiers 5.4

MISRA-C Rule 5.4 Description: A tag name shall be a unique identifier.

This rule is not included in Mü8004 standard. Although Mü8004 Rule 0.2.9 establishes that same variables should not serve different purposes and it would be useful to avoid the reuse of names within a program for same purposes. This would be useful adding it to avoid confusion.

MISRA –C Identifiers 5.5

MISRA-C Rule 5.5 Description: No object or function identifier with static storage duration should be reused.

This rule is not included in Mü8004 standard. It would be useful adding it because the possibility exists for the user to incorrectly associate unrelated variables with the same name.

MISRA –C Identifiers 5.6

MISRA-C Rule 5.6 Description: No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure member and union member names.

This rule is not included in Mü8004 standard. It extends the avoidance of using same names for same or different purposes. It could be an advisory request to avoid confusion.

MISRA –C Identifiers 5.7

MISRA-C Rule 5.7 Description: No identifier name should be reused (across any files in the system).

This rule is not included in Mü8004 standard. It incorporated the Rules 5.2, 5.3, 5.4, 5.5 and 5.6. It would be an extremely severe requirement to avoid confusion between names.

Mü8004 Identifiers 0.2.9

Some points of this section, as the following ones, are very important to avoid confusion between names and are not included in Identifiers section in MISRA-C standard:

- Uppercase and lowercase letters, numbers from 0 to 9, and the sign $ and _ are allowed for defining names. '_' must not be the first character
- Identifiers that differ only in uppercase/lowercase characters must not have different meaning
- Identifiers for macros shall be written in uppercase letter

TYPES:

MISRA –C Types 6.1

MISRA-C Rule 6.1 Description: The plain char type shall be used only for the storage and use of character values

This rule is not included in Mü8004 standard. This rule could be useful to set the restriction in the work with this type of data.

MISRA –C Types 6.2

MISRA-C Rule 6.2 Description: signed and unsigned char type shall be used only for the storage and use of numeric values. Plain char type shall be used for character data.

This rule is not included in Mü8004 standard. This rule could be useful to make a difference between the work with numeric and character data.

MISRA –C Types 6.3

MISRA-C Rule 6.3 Description: Typedefs that indicate size and signedness should be used in place of the basic numerical types

Mü8004 standard does not show how to use typedef with different data types. However, it shows in Rule 0.2.6 that float and double data types are not supported. This could be a drawback for the precision of variables and operations between them.

MISRA –C Types 6.4

MISRA-C Rule 6.4 Description: Bit fields shall only be defined to be of type unsigned int or signed int.

This rule is not included in Mü8004 standard, although Mü8004 Rule 0.2.6 establishes that bitfields are permitted. This rule could be useful for the correctness in the work with bitfields.

MISRA –C Types 6.5

MISRA-C Rule 6.5 Description: Bit fields of signed type shall be at least 2 bits long.

This rule is not included in Mü8004 standard

CONSTANTS:

MISRA –C Types 7.1

MISRA-C Rule 7.1 Description: Octal constants (other than zero) and octal escape sequences shall not be used

This rule is not included in Mü8004 standard for the definition of constants. Although it defines how to create constants using “const” keyword, it could be interesting not to mix constants and octal, because of potential errors when working with fixed length constants

Mü8004 – 0.2.10 Constants

Mü8004 – 0.2.10 Constants Rule: Although MISRA-C adds 7.1 Rule that describes the work with octal constants in the standard, Mü8004 - 0.2.10 Constants rule specifies better which is the way to define constants of different type, and the way to use them along the code.

DECLARATIONS AND DEFINITIONS:

MISRA –C Types 8.1

MISRA-C Rule 7.1 Description: Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call

Mü8004 standard includes at Rule 0.2.15 that function prototypes shall be used for every function whenever the compiler supports it. It would be useful to set the visibility of prototypes for the integrity of function definitions and calls.

MISRA –C Declarations and Definitions 8.2

MISRA-C Rule 8.2 Description: Whenever an object or function is declared or defined, its type shall be explicitly stated

Mü8004 standard includes at Rule 0.2.15 that function header have to define the function type.

MISRA –C Declarations and Definitions 8.3

MISRA-C Rule 8.3 Description: For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical

Mü8004 standard does not include this rule, but this rule should be necessary for the proper operation of the function.

MISRA –C Declarations and Definitions 8.4

MISRA-C Rule 8.4 Description: If objects or functions are declared more than once their types shall be compatible

Mü8004 standard does not include this rule. It might be good to add this rule to the proper functioning of the code, even though it would be recommendable not to declare objects or functions more than once in order to reduce the number of mistakes made.

MISRA –C Declarations and Definitions 8.5

MISRA-C Rule 8.5 Description: There shall be no definitions of objects or functions in a header file

Mü8004 standard does not include this rule. Although Mü8004 Rule 0.2.5 establishes that function prototypes shall only be used in header files, there is no reference to the definition of objects and functions. To prohibit the definition of objects and functions in the header file would be a good programming rule.

MISRA –C Declarations and Definitions 8.6

MISRA-C Rule 8.6 Description: Functions shall be declared at file scope

There is a restriction in Mü8004 standard Rule 0.2.5 when declaring functions at file scope, because it establishes that function prototypes shall only be used in header files.

MISRA –C Declarations and Definitions 8.7

MISRA-C Rule 8.7 Description: Objects shall be defined at block scope if they are only accessed from within a single function

Mü8004 standard Rule 0.2.14 establishes that global definitions of structures shall be defined in header files. It would be interesting to add MISRA-C Rule 8.7 for objects that are only used in functions or in block scope.

MISRA –C Declarations and Definitions 8.8

MISRA-C Rule 8.8 Description: An external object or function shall be declared in one and only one (external) file

This rule can be added to Mü8004 standard to improve 0.2.14 Rule, as it defines that global definition of structures shall be defined in header files

MISRA –C Declarations and Definitions 8.9

MISRA-C Rule 8.9 Description: An identifier with external linkage shall have exactly one external definition

Mü8004 does not include this rule. It is necessary to add this rule to fix the work with extern parameter.

MISRA –C Declarations and Definitions 8.10, 8.11

MISRA-C Rule 8.10 Description: All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required

MISRA-C Rule8.11 Description: The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage

Mü8004 does not include these rules. It would be useful to avoid confusion between objects with internal scope and objects with external scope.

MISRA –C Declarations and Definitions 8.12

MISRA-C Rule 8.12 Description: When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization

Mü8004 does not include this rule. It could be interesting to add this rule to Mü8004 standard to establish the work with arrays when they are declared with external linkage.

INITIALIZATION:

MISRA –C Initialization 9.1

MISRA-C Rule 9.1 Description: All automatic variables shall have been assigned a value before being used

Mü8004 standard includes at Rule 0.2.12 that all variables and fields shall be initialized before they are used the first time.

MISRA –C Initialization 9.2

MISRA-C Rule 9.2 Description: Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures

The use of braces is included in Mü8004 standard Rule 0.2.6, as it establishes that for clearness, the initialization shall be put in curly brackets.

MISRA –C Initialization 9.3

MISRA-C Rule 9.3 Description: In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized

Mü8004 does not include this rule. It is necessary to avoid making mistakes when initializing enumerators.

CONVERSIONS:

MISRA -C Conversions 10.1

MISRA-C Rule 10.1 Description: The value of an expression of integer type shall not be implicitly converted to a different underlying type if:
(a) it is not a conversion to a wider integer type of the same signedness, or
(b) the expression is complex, or

(c) the expression is not constant and is a function argument, or
(d) the expression is not constant and is a return expression.

Mü8004 includes this rule. It is a precision of the third phrase of the chapter 0.2.18.

MISRA -C Conversions 10.2

MISRA-C Rule 10.2 Description: The value of an expression of floating type shall not be implicitly converted to a different type if:
(a) it is not a conversion to a wider floating type, or
(b) the expression is complex, or
(c) the expression is a function argument, or
(d) the expression is a return expression.

Mü8004 includes this rule. It is a precision of the thrid phrase of the chapter 0.2.18.

MISRA -C Conversions 10.3

MISRA-C Rule 10.3 Description: The value of a complex expression of integer type shall only be cast to a type of the same signedness that is no wider than the underlying type of the expression.

Mü8004 includes this rule. It is a precision of the third sentence of the chapter 0.2.18.

MISRA -C Conversions 10.4

MISRA-C Rule 10.4 Description: The value of a complex expression of floating type shall only be cast to a floating type that is narrower or of the same size.

Mü8004 includes this rule. It is a precision of the third sentence of the chapter 0.2.18.

MISRA -C Conversions 10.5

MISRA-C Rule 10.4 Description: If the bitwise operators ãnd << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Mü8004 does not include this rule. This operators converts the data in the type having a small size. This convertion causes overflow and bug.

MISRA -C Conversions 10.6

MISRA-C Rule 10.6 Description: A “U” suffix shall be applied to all constants of unsigned type.

Mü8004 does not include this rule. This rule is useful for the maintenability and code review.

MISRA -C Conversions 11.1

MISRA-C Rule 11.1 Description: Conversions shall not be performed between a pointer to a function and any type other than an integral type.

Mü8004 does not include this rule. If this rule is not respected, the code has an undefined behavior. This rule must be mandatory. In addition, the code becomes independent of the compiler.

MISRA -C Conversions 11.2

MISRA-C Rule 11.2 Description: Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.

Mü8004 does not include this rule. This rule is for the determinism of the cast system. This rules indicates that we do not have to mix pointer and object.

MISRA -C Conversions 11.3

MISRA-C Rule 11.3 Description: A cast should not be performed between a pointer type and an integral type.

Mü8004 does not include this rule. This rule is used for not mixing data and pointer. Those two objects are different.

MISRA -C Conversions 11.4

MISRA-C Rule 11.4 Description: A cast should not be performed between a pointer to object type and a different pointer to object type.

Mü8004 does not include this rule. This rule is used for not mixing different pointer types for data alignement. This rule allows to execute the code in a safety way.

MISRA -C Conversions 11.5

MISRA-C Rule 11.5 Description: A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer.

Mü8004 does not include this rule. This rule is used for not modifying a constant data or a volatile. This rule allows to execute the code in a safety way.

EXPRESSIONS:

MISRA –C Expressions 12.1

MISRA-C Rule 12.1 Description: Limited dependence should be placed on C's operator precedence rules in expressions

Mü8004 does not include this rule. This could be a good advisory rule for the developer that has to be careful with made mistakes because of precedence rule of C. Parentheses should be used to reduce made mistakes.

MISRA –C Expressions 12.2

MISRA-C Rule 12.2 Description: The value of an expression shall be the same under any order of evaluation that the standard permits

Mü8004 standard makes reference to the importance of the influence of evaluation order in expressions. That's why Mü8004 Rule 0.2.13 establishes that assignments inside expressions are forbidden and Mü8004 Rule12.2 establishes that post increment and post decrement operators are only allowed if they are placed in a separate expression. However, an advice should be made to the influence of access order in expressions where functions calls, access to volatile objects... are used.

MISRA –C Expressions 12.3

MISRA-C Rule 12.3 Description: The sizeof operator shall not be used on expressions that contain side effects

Mü8004 Rule 0.2.4 establishes that sizeof operator must not be used after #if. This operator cannot be used to evaluate an expression. It shall only be applied to an operand which is a type or object.

MISRA –C Expressions 12.4

MISRA-C Rule 12.4 Description: The right-hand operand of a logical && or || operator shall not contain side effects

Mü8004 standard does not include this rule. This could be a good rule for the developer that has to be careful with side effects when working with these operators.

MISRA –C Expressions 12.5

MISRA-C Rule 12.5 Description: The operands of a logical && or || shall be primary-expressions Mü8004 standard does not include this rule. This could be a good rule for both readability of code and for ensuring that the behavior is as the programmer intended.

MISRA –C Expressions 12.6

MISRA-C Rule 12.6 Description: The operands of logical operators (&&, || and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, ||, !, =, ==, != and ?:)

Mü8004 Rule 0.2.4 establishes the difference between logical operators (&&, ||, !, =, ==, != and ?:) and bitwise (&=, |, ¸ -, », «) operators

MISRA –C Expressions 12.7

MISRA-C Rule 12.7 Description: Bitwise operators shall not be applied to operands whose underlying type is signed

Mü8004 Rule 0.2.4 establishes that bitwise operators and right shift operators shall only be used with unsigned variables.

MISRA –C Expressions 12.8

MISRA-C Rule 12.8 Description: The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand

Mü8004 standard does not include this rule. It could be useful to add this rule and others that talk about the limitations in the work with different operands.

MISRA –C Expressions 12.9

MISRA-C Rule 12.9 Description: The unary minus operator shall not be applied to an expression whose underlying type is unsigned

Mü8004 Rule 0.2.6 explains the problematic of combining signed and unsigned variables in arithmetic operations. However, this should be extended to explain the problems generated when doing operations as applying operators like unary minus to unsigned variables.

MISRA –C Expressions 12.10

MISRA-C Rule 12.10 Description: The comma operator shall not be used

Mü8004 Rule 0.2.4 establishes that Mü8004 standard does not support the work with comma operator.

MISRA –C Expressions 12.11

MISRA-C Rule 12.11 Description: Evaluation of constant unsigned integer expressions should not lead to wrap-around

Mü8004 standard does not include this rule. This could be a helpful rule to avoid the overflow of unsigned integer expressions.

MISRA –C Expressions 12.12

MISRA-C Rule 12.12 Description: The underlying bit representations of floating-point values shall not be used

Mü8004 standard does not include this rule. This could be an interesting rule to avoid the errors caused by the way floating-point values are stored, in case this data types would be supported by the compiler.

MISRA –C Expressions 12.13

MISRA-C Rule 12.13 Description: The increment (++) and decrement (–) operators should not be mixed with other operators in an expression

Mü8004 Rule 0.2.4 establishes the restrictions when working with these operators. Post increment and post decrement operators are only allowed if they are placed in a separate expression.

CONTROL STATEMENT EXPRESSIONS:

MISRA –C Control statement expressions 13.1

MISRA-C Rule 13.1 Description: Assignment operators shall not be used in expressions that yield a Boolean value

Mü8004 Rule 0.2.13 establishes that assignment operators shall not be used inside expressions that are considered to have a Boolean value.

MISRA –C Control statement expressions 13.2

MISRA-C Rule 13.2 Description: Tests of a value against cero should be made explicit, unless the operand is effectively Boolean

Mü8004 standard does not include this rule. It could be useful to add this rule for the appropriate work with “not equal” operator.

MISRA –C Control statement expressions 13.3

MISRA-C Rule 13.3 Description: Floating-point expressions shall not be tested for equality or inequality

Mü8004 standard does not include this rule. This could be an interesting rule if floating-point values are allowed.

MISRA –C Control statement expressions 13.4

MISRA-C Rule 13.4 Description: The controlling expression of a for statement shall not contain any object of floating type

Mü8004 standard does not include this rule. This could be an interesting rule to avoid making mistakes with for statement, if floating-point values are allowed.

MISRA –C Control statement expressions 13.5

MISRA-C Rule 13.5 Description: The three expressions of a for statement shall be concerned only with loop control

Mü8004 standard does not include this rule. This is a necessary rule that explains how to work correctly with for statement.

MISRA –C Control statement expressions 13.6

MISRA-C Rule 13.6 Description: Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop

Mü8004 standard does not include this rule. This is a basic rule for the correct work of for loop.

MISRA –C Control statement expressions 13.7

MISRA-C Rule 13.7 Description: Boolean operations whose results are invariant shall not be permitted

Mü8004 standard does not include this rule. This could be a good rule to avoid the propagation of errors in the program due to wrongly implemented Boolean operations.

CONTROL FLOW:

MISRA –C Control flow 14.1

MISRA-C Rule 14.1 Description: There shall be no unreachable code

Mü8004 standard does not include this rule, but it is a necessary to avoid mistakes due to code that it is never executed.

MISRA –C Control flow 14.2

MISRA-C Rule 14.2 Description: All non-null statements shall either: (a) have at least one side-effect however executed, or (b) cause control flow to change

Mü8004 standard does not include this rule. This is a necessary rule to avoid making errors when creating statements.

MISRA –C Control flow 14.3

MISRA-C Rule 14.3 Description: 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

Mü8004 standard does not include this rule. This is a necessary rule if null statements are allowed to be used. However, the safest way would be not to permit embedding null statements in the code.

MISRA –C Control flow 14.4

MISRA-C Rule 14.4 Description: The *goto* statement shall not be used

Mü8004 standard does not include this statement in the list of permitted statements.

MISRA –C Control flow 14.5

MISRA-C Rule 14.5 Description: The *continue* statement shall not be used

Mü8004 Rule 0.2.3 includes this statement in the list of permitted statements, even though it is recommended to avoid working with it, if possible.

MISRA –C Control flow 14.6

MISRA-C Rule 14.6 Description: For any iteration statement there shall be at most one break statement used for loop termination

Mü8004 Rule 0.2.3 establishes that every case branch must contain a statement and end with break. This rule is in the interest of good structured programming.

MISRA –C Control flow 14.7

MISRA-C Rule 14.7 Description: A function shall have a single point of exit at the end of the function

Mü8004 Rule 0.2.3 includes this restriction, as it establishes the use, once per function, of the return statement as the exit point of the function.

MISRA –C Control flow 14.8

MISRA-C Rule 14.8 Description: The statement forming the body of a *switch, while, do ... while* or *for* statement shall be a compound statement.

Mü8004 Rule 0.2.3 establishes that to facilitate the examination, the program shall be structured with brackets and indentation of lines. This rule should be extended to mention specific cases as, *switch, while, do ... while* and *for* cases.

MISRA –C Control flow 14.9

MISRA-C Rule 14.9 Description: An *if* (expression) construct shall be followed by a compound statement. The *else* keyword shall be followed by either a compound statement, or another *if* statement

Mü8004 Rule 0.2.3 defines the construction of *if* expression. However, it is less restrictive as for an *if* expression with a single statement, braces are not required.

MISRA –C Control flow 14.10

MISRA-C Rule 14.10 Description: All *if... else if* construct shall be terminated with an else clause

Mü8004 Rule 0.2.3 defines the construction of *if* expression. However, it does not establish how it is the work with this advanced structure.

SWITCH STATEMENTS:

MISRA –C Switch Statement 15.0

MISRA-C Rule 15.0 Description: The MISRA C *switch* syntax shall be used

Mü8004 Rule 0.2.3 includes, in a less detailed way, how the construction of a *switch* statement is.

MISRA –C Switch Statement 15.1

MISRA-C Rule 15.1 Description: A *switch* label shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement

Mü8004 Rule 0.2.3 includes how *switch, case* and *default* labels have to be used in a *switch* statement.

MISRA –C Switch Statement 15.2

MISRA-C Rule 15.2 Description: An unconditional *break* statement shall terminate every non-empty *switch* clause

Mü8004 Rule 0.2.3 establishes that every case branch in a *switch* statement must contain a statement and end with *break*.

MISRA –C Switch Statement 15.3

MISRA-C Rule 15.3 Description: The final clause of a *switch* statement shall be the default clause

Mü8004 Rule 0.2.3 establishes that a default case must be defined in a *switch* statement, and that this is the last statement in the *switch* block.

MISRA –C Switch Statement 15.4

MISRA-C Rule 15.4 Description: A *switch* expression shall not represent a value that is effectively Boolean

Mü8004 standard does not include this rule. It could be a useful rule to know which data types can be used with *switch* statement, and avoid making mistakes when the *switch* statement is

MISRA –C Switch Statement 15.5

MISRA-C Rule 15.5 Description: Every *switch* statement shall have at least one *case* clause

Mü8004 standard does not include this rule. It could be useful to add this rule to clarify the necessity of adding at least one case clause in every *switch* statement.

FUNCTION:

MISRA -C Rule 16.1

MISRA -C Rule 16.1 Description : Functions shall not be defined with a variable number of arguments.

This rule is included in Mü8004 standard. This rule is mandatory. A variable number of arguments is dangerous for the code. It can introduce undefined behavior.

MISRA -C Rule 16.2

MISRA -C Rule 16.2 Description : Functions shall not call themselves, either directly or indirectly.

This rule is included in Mü8004 standard. This rule is mandatory. The functions who call themselves, introduce a recursive behavior. And with recursive functions, it is difficult to justify the program stop.

MISRA -C Rule 16.3

MISRA -C Rule 16.3 Description : Identifiers shall be given for all of the parameters in a function prototype declaration.

This rule is not included in Mü8004 standard. This rule is used for compatibility, clarity and maintainability aspects.

MISRA -C Rule 16.4

MISRA -C Rule 16.4 Description : The identifiers used in the declaration and definition of a function shall be identical.

This rule is not included in Mü8004 standard. This rule is useful for the coherency between the c file and the h file.

MISRA -C Rule 16.5

MISRA -C Rule 16.5 Description : Functions with no parameters shall be declared and defined with the parameter list void.

This rule is not included in Mü8004 standard. This rule is used for standardizing the list parameter and the return of the function.

MISRA -C Rule 16.6

MISRA -C Rule 16.5 Description : The number of arguments passed to a function shall match the number of parameters.

This rule is not included in Mü8004 standard. This rule is related to the rule 16.1. This rule is used for standardizing the definition and the call function.

MISRA -C Rule 16.7

MISRA -C Rule 16.7 Description : A pointer parameter in a function prototype should be declared as pointer to *const* if the pointer is not used to modify the adressed object.

This rule is not included in Mü8004 standard. This rule is mandatory. Thanks to this rule verifications can be made by the compiler.

MISRA -C Rule 16.8

MISRA -C Rule 16.8 Description : All exit paths from a function with non-void return type shall have an explicit *return* statement with an expression.

This rule is not included in Mü8004 standard. The C norm does not define the behaviour when there is no expression in the dataflow. This rule prevents an unwanted behavior.

MISRA -C Rule 16.9

MISRA -C Rule 16.9 Description : A function identifier shall only be used with either a preceding &,or with a parenthesised parameter list, which may be empty.

This rule is not included in Mü8004 standard. In the example of Misra, we cannot make the difference between the null test of a pointer or the return of the function. To avoid a confusion, this rule is mandatory.

MISRA -C Rule 16.9

MISRA -C Rule 16.9 Description : A function identifier shall only be used with either a preceding &,or with a parenthesised parameter list, which may be empty.

This rule is not included in Mü8004 standard. This rule is obvious. However the rule 20.3 indicates that a test before the call function is recommended.

We have noticed that there was no many information about the behavior and the function call in this standard, but a lot of information on the documentation and the declaration of the function.

POINTERS AND ARRAYS:

MISRA –C Pointers and Arrays 17.1

MISRA-C Rule 17.1 Description: Pointer arithmetic shall only be applied to pointers that address an array or array element

Mü8004 standard does not include this rule. This is a necessary rule to determine how pointers arithmetic has to be applied in order to have an expected behaviour.

MISRA –C Pointers and Arrays 17.2

MISRA-C Rule 17.2 Description: Pointer subtraction shall only be applied to pointers that address elements of the same array

Mü8004 standard does not include this rule. This is a necessary rule if the result we want to get is the number of elements separating the pointers.

MISRA –C Pointers and Arrays 17.3

MISRA-C Rule 17.3 Description: >, >=, <, <= shall not be applied to pointer types except where they point to the same array

Mü8004 standard does not include this rule. This is a necessary rule if the behavior we want to obtain after the comparison of the pointers it is a well-defined behavior.

MISRA –C Pointers and Arrays 17.4

MISRA-C Rule 17.4 Description: Array indexing shall be the only allowed form of pointer arithmetic

Mü8004 standard does not include this rule. This rule would help to avoid making mistakes like accessing to invalid memory addresses after manipulation of pointers.

MISRA –C Pointers and Arrays 17.5

MISRA-C Rule 17.5 Description: The declaration of objects should contain no more than 2 levels of pointer indirection

Mü8004 standard does not include this rule. Although this would not be a required rule, it would help to improve the readability of the code and avoid making mistakes because of the complexity of instructions.

MISRA –C Pointers and Arrays 17.6

MISRA-C Rule 17.6 Description: The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist

Mü8004 Rule 0.2.17 does include this rule, as it establishes that addresses of auto variables shall only be stored in auto variables of the same visibility.

Mü8004 – 0.2.17 Pointer

Mü8004 – 0.2.17 Pointer Rule: Although MISRA-C includes rules about Pointers, it is necessary to establish the limitation of the relation between pointers and the functions, and pointers and the definition of some variables.

STRUCTURES AND UNIONS:

MISRA –C Structures and Unions 18.1

MISRA-C Rule 18.1 Description: All structure and union types shall be complete at the end of a translation unit

Mü8004 standard does not include this rule. This is a basic rule that shows how the definition of structures has to be made.

MISRA –C Structures and Unions 18.2

MISRA-C Rule 18.2 Description: An object shall not be assigned to an overlapping object

Mü8004 standard does not include this rule. Although this rule refers to low-level programming, it could be useful in case it is permitted to create two objects having some overlap in memory.

MISRA –C Structures and Unions 18.3

MISRA-C Rule 18.3 Description: An area of memory shall not be reused for unrelated purposes

Mü8004 standard does not include this rule. This could be an interesting rule to avoid making mistakes by storing unrelated data in the same piece of memory. However, exceptions should be made for requirements of memory efficiency.

MISRA –C Structures and Unions 18.4

MISRA-C Rule 18.4 Description: Unions shall not be used

Mü8004 Rule 0.2.6 contradicts this rule. Unions could be used in situations in which the use of unions is advisable for an implementation that has to be efficient in terms of memory.

PREPROCESSING DIRECTIVES:

MISRA -C Rule 19.1

MISRA -C Rule 19.1 Description : #include statements in a file should only be preceded by other preprocessor directives or comments.

This rule is included in Mü8004. This rule is advisory in MISRA -C but mandatory in the Mü8004. This rule describes the pattern of the files and makes the files more readable.

MISRA -C Rule 19.2

MISRA -C Rule 19.3 Description : Non-standard characters should not occur in header file names in #include directives.

This rule is not included in Mü8004. This rule is useful for the portability of the code.

MISRA -C Rule 19.3

MISRA -C Rule 19.3 Description : The #include directive shall be followed by either a <filename> or "filename" sequence.

This rule is included in Mü8004. This rule is used to demonstrate the dependence of the modular code.

MISRA -C Rule 19.4

MISRA -C Rule 19.4 Description : C macros shall only expand to a braced initialiser, a constant, a string literal, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

This rule is not included in Mü8004. This rule reduces C macros. This reduction is useful for the maintainability and readability of the code. This rule is used only to define macro instructions and not a function.

MISRA -C Rule 19.5

MISRA -C Rule 19.5 Description : Macros shall not be #define'd or #undef'd within a block.

This rule is not included in Mü8004. This rule is not reduce the scope of the macro in a code. When it defines a macro, the scope of the instruction must be at same level.

MISRA -C Rule 19.6

MISRA -C Rule 19.6 Description : #undef shall not be used.

This rule is included in Mü8004. #undef is no used. And this instruction can lead to confusion.

MISRA -C Rule 19.7

MISRA -C Rule 19.7 Description : A function should be used in preference to a function-like macro

This rule is not included in Mü8004. This rule is used to not confuse the modular aspect and the programming.

MISRA -C Rule 19.8

MISRA -C Rule 19.8 Description : A function-like macro shall not be invoked without all of its arguments.

This rule is not included in Mü8004. This rule is mandatory. This rule is used to avoid the unwanted behaviour.

MISRA -C Rule 19.9

MISRA -C Rule 19.9 Description : Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

This rule is not included in Mü8004. This rule is for readable. If this rule is not respected, the software maintainer will be allowed to confuse the macro and preprocessing directive?

MISRA -C Rule 19.10

MISRA -C Rule 19.10 Description : In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.

This rule is not included in Mü8004. This rule is used to create the unwanted behaviour, when the pre-processor instances the macro.

MISRA -C Rule 19.11

MISRA -C Rule 19.11 Description : All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.

This rule is not included in Mü8004. This rule is for the modular aspect and comprehension. Because The identifier is clearly defined.

MISRA -C Rule 19.12

MISRA -C Rule 19.12 Description : There shall be at most one occurrence of the # or ## operators in a single macro definition.

This rule is not included in Mü8004. This rule is used to avoid unspecified directive preprocessor.

MISRA -C Rule 19.13

MISRA -C Rule 19.13 Description : The # and ## operators should not be used.

This rule is not included in Mü8004. This rule is used to avoid unspecified directive preprocessor.

MISRA -C Rule 19.14

MISRA -C Rule 19.14 Description : The defined preprocessor operator shall only be used in one of the two standard forms.

This rule is not included in Mü8004. This rule is useful for the portability of the code.

MISRA -C Rule 19.15

MISRA -C Rule 19.15 Description : Precautions shall be taken in order to prevent the contents of a header file being included twice.

This rule is not included in Mü8004. This rule is used to avoid incoherence file inclusion.

MISRA -C Rule 19.16

MISRA -C Rule 19.16 Description : Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

This rule is not included in Mü8004. This rule is here to avoid a breach of C norm. If prepocessing directives is badly formed, the compiler will ignor it without warning.

MISRA -C Rule 19.17

MISRA -C Rule 19.17 Description : All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

This rule is not included in Mü8004. This rule is used for the maintenance of the code because the imbrication of preprocessor directive makes the code hard to read.

STANDARD LIBRARIES:

MISRA -C Rule 20.1

MISRA -C Rule 20.1 Description : Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.

This rule is not included in Mü8004. This rule is both for the standardization and readability of the code.

MISRA -C Rule 20.2

MISRA -C Rule 20.2 Description : The names of standard library macros, objects and functions shall not be reused.

This rule is not included in Mü8004. This rule is both for standardization and readability of the code.

MISRA -C Rule 20.3

MISRA -C Rule 20.3 Description : The validity of values passed to library functions shall be checked.

This rule is included in Mü8004. This rule is mandatory. When a standard function is called , the values passed to the function are checked. This rule is important because it avoids unwanted behaviour.

MISRA -C Rule 20.4

MISRA -C Rule 20.4 Description : Dynamic heap memory allocation shall not be used.

This rule is included in Mü8004. Dynamic heap memory utilisation is a source of errors such as memory leak or null pointer.

MISRA -C Rule 20.5

MISRA -C Rule 20.5 Description : The error indicator *errno* shall not be used.

This rule is not included in Mü8004. In the standard *errno* is poorly defined. To avoid unwanted behaviour this construction must be prohibited.

paragraphMISRA -C Rule 20.6 MISRA -C Rule 20.6 Description : The macro *offsetof*, in library <*stddef.h*>, shall not be used.

This rule is not included in Mü8004. In the standard the macro offsetof is bad defined. To avoid unwanted behaviour this construction must be prohibited.

MISRA -C Rule 20.7

MISRA -C Rule 20.7 Description : The *setjmp* macro and the *longjmp* function shall not be used.

This rule is included in Mü8004. The utilisation of this instruction avoid to use goto and label. This instructions provide "spaghetti code".

MISRA -C Rule 20.8

MISRA -C Rule 20.8 Description : The signal handling facilities of <*signal.h*> shall not be used.

This rule is not included in Mü8004. The behaviour of <*signal.h*> is undefined for some values. And a critical code must not have undefined behavior.

MISRA -C Rule 20.9

MISRA -C Rule 20.9 Description : The input/output library <*stdio.h*> shall not be used in production code.

This rule is not included in Mü8004. The behaviour of <*stdio.h*> is undefined for some values. And a critical code must not have undefined behavior.

MISRA -C Rule 20.10

MISRA -C Rule 20.10 Description : The library functions *atof*, *atoi* and *atol* from library <*stdlib.h*> shall not be used.

This rule is not included in Mü8004. In the standard the macro atof, atoi and atol are bad defined. To avoid unwanted behaviour this construction shall be prohibited.

MISRA -C Rule 20.11

MISRA -C Rule 20.11 Description : The library functions *abort*, *exit*, *getenv* and *system* from library <*stdlib.h*> shall not be used.

This rule is not included in Mü8004. This function are not need in embedded code. Therefore it is forbidden.

MISRA -C Rule 20.12

MISRA -C Rule 20.12 Description : The time handling functions of library <*time.h*> shall not be used.

This rule is not included in Mü8004. The behaviour of <time.h> is undefined for some values. And a critical code must not have undefined behavior.

DOCUMENTATION:

MISRA –C Documentation 3.1

MISRA-C Rule 3.1 Description: All usage of implementation-defined behavior shall be documented

Mü8004 standard does not include this rule. This could be a useful rule to guarantee that the standard's behavior is completely documented and covered by the defined rules.

MISRA –C Documentation 3.2

MISRA-C Rule 3.2 Description: The character set and the corresponding encoding shall be documented

Mü8004 standard does not include this rule. As standard's requirements have to be documented, same thing should be made with encoding of permitted character sets.

MISRA –C Documentation 3.3

MISRA-C Rule 3.3 Description: The implementation of integer division in the chosen compiler should be determined, documented and taken into account

Mü8004 standard does not include this rule. It should be documented the way arithmetic operations are done, and what are the limitations of operators and the expected behavior.

MISRA –C Documentation 3.4

MISRA-C Rule 3.4 Description: All uses of *#pragma* directive shall be documented and explained

Mü8004 standard does not include this rule. Although Mü8004 Rule 0.2.5 establishes that the use of the *#pragma* command requires a special explanation in the proof of functionality, it doesn't require documenting its use.

MISRA –C Documentation 3.5

MISRA-C Rule 3.5 Description: If it is being relied upon, the implementation defined behavior and packing of bitfields shall be documented

Mü8004 standard does not include this rule. It could be a useful rule to settle how the work with bit fields has to be done.

MISRA –C Documentation 3.6

MISRA-C Rule 3.6 Description: All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation

Mü8004 standard does not include this rule. It could be a useful rule to document what libraries have been used in the production of the code, or the ones supplied by the compiler.

Mü8004 – 0.2.3 Coding of Basic Structures

Mü8004 – 0.2.3 Coding of Basic Structures Rule: Although MISRA-standard includes the way of working with basic structures like if-else, switch-case and do-while, Mü8004 standard defines clearly how this structures have to be defined.

Mü8004 – 0.2.4 Operators

Mü8004 – 0.2.4 Operator Rule: Although MISRA-standard includes explanation for the most important operators, it is helpful to have a general overview of them within a table.

Mü8004 - 0.2.5 Preprocessor Commands

This rule is used to the portability of the code. #pragma directive is for the optimization compiler, and often it is not portable. This commands must be forbid.

Mü8004 - 0.2.7 Memory Classes

The permitted memory classes is classical. But the utilisation of static is problematic because the code is unreadable.

Mü8004 – 0.2.11 Variables

Mü8004 – 0.2.11 Variables Rule: Mü8004 - 0.2.11 Variables rule explains the correct way of defining variables. Although the content of this rule has been treated in MISRA-C standard, it is appropriate to use a specific section to explain the work with variables.

Mü8004 0.2.14 Sources Files

This rule is more restrictive than MISRA. The pattern of Mü8004 describes all the file.

Mü8004 – 0.2.19 Data References

Mü8004 – 0.2.19 Data References Rule: Although MISRA-C includes rules about Documentation, it is necessary to make a reference to the documentation of data related and not related to the project planning, and not only to the data and to the libraries used along the code.

Mü8004 – 0.2.20 Cross Reference List

Mü8004 – 0.2.20 Cross Reference List Rule: MISRA-C doesn't include the necessity of using cross reference list for the data of the code. It also could be useful to add the characteristics that need to have the development tools used for this purpose.

Mü8004 – 0.2.21 Assembler Coding

Mü8004 – 0.2.21 Assembler Coding Rule: MISRA-C doesn't include the necessity of justifying the use of assembler coding. It is helpful to specifying that assembler could be useful in time critical programming.

Mü8004 - 0.2.23 Libraries Routine

This section defined some restrictive utilisation of the standard library. But This restriction is less restrictive than MISRA.

Mü8004 - 0.2.24 Program Protocol

this rule is obvious but it is not delivered of metric in the code for readable, the maximum number of line in the function or in the file ...

Mü8004 – 0.2.25 Optimization

Mü8004 – 0.2.25 Optimization Rule: Although MISRA-standard includes the importance of the characteristics of compiler used in the development environment, it is useful to explain the influence of optimization in the compiler.

4.9 Conclusions

Static analysis tools are very good due to the detection of several problem/errors at code level that are usually difficult to detect by manual inspection. Furthermore, they help enforce coding standards and keep code complexity low.

However, these tools sometimes report false positives so it is necessary review them and decide if they are related with problems or not. Nonetheless, it is recommended to complement the static analysis tools with manual code inspections (not thought of by the original coder) and dynamic analysis.

In order to ensure the correctness of the obtained results mentioned in the previous sections, a comparison of them was executed.

Table 4.28. File Size metrics comparation

Bitwalker.c				
Metric	**RSM**	**LocMetric**	**Understand**	**CMT++**
Total lines:	109	110	109	109
Code/program lines:	58	58	58	58
Comment lines:	29	28+5 = 33	33	33
Blank lines:	-	24	23	23

As a result of this comparison we obtain that between the tools there are some small deviations regarding some code metrics like total lines, comments or blank lines. Thus it was necessary to check how each aspect/metric is defined into each tool.

A manual inspection was done and the source of inconsistency is due to LocMetricss counts the last blank line of the file and RSM tool does not count the blank lines that are inside one commented section.

In addition to code size metrics of file, size code metrics per function were compared.

Table 4.29. Functions Size metrics comparation

Metric	**RSM**	**LocMetric**	**Understand**	**CMT++**
Bitwalker_Peek				
Total lines:	19	-	20	20
Code/program lines:	12	-	13	13
Comment lines:	5	-	5	5
Blank lines:	-	-	4	4
Bitwalker_Poke				
Total lines:	23	-	24	24
Code/program lines:	16	-	17	17
Comment lines:	6	-	6	6
Blank lines:	-	-	4	4
Bitwalker_IncrementalWalker_Init				
Total lines:	5	-	6	6
Code/program lines:	5	-	6	6
Comment lines:	0	-	0	0
Blank lines:	-	-	0	0
Bitwalker_IncrementalWalker_Peek_Next				
Total lines:	6	-	7	7
Code/program lines:	5	-	6	6
Comment lines:	1	-	1	1
Blank lines:	-	-	0	0

Table 4.29. Functions Size metrics comparation

Bitwalker_IncrementalWalker_Peek_Finish				
Total lines:	3	-	4	4
Code/program lines:	3	-	4	4
Comment lines:	0	-	0	0
Blank lines:	-	-	0	0
Bitwalker_IncrementalWalker_Poke_Next				
Total lines:	6	-	7	7
Code/program lines:	5	-	6	6
Comment lines:	1	-	1	1
Blank lines:	-	-	0	0
Bitwalker_IncrementalWalker_Poke_Finish				
Total lines:	3	-	4	4
Code/program lines:	3	-	4	4
Comment lines:	0	-	0	0
Blank lines:	-	-	0	0

The total lines and program lines counts produced by some of the tool for the same product differ a little bit. The results clearly demonstrate the effects of existing ambiguities in code counting methodology and a variety of interpretations.

McCabe cyclomatic complexity was another complexity metric calculated by different of the selected tools.

Table 4.30. function Cyclomatic Complexity comparation

Function	RSM	Understand	CMT++
Bitwalker_Peek	3	3	3
Bitwalker_Poke	5	5	5
Bitwalker_IncrementalWalker_Init	1	1	1
Bitwalker_IncrementalWalker_Peek_Next	1	1	1
Bitwalker_IncrementalWalker_Peek_Finish	1	1	1
Bitwalker_IncrementalWalker_Poke_Next	1	1	1
Bitwalker_IncrementalWalker_Poke_Finish	1	1	1

According to McCabe a value of 10 is a practical upper limit for the cyclomatic complexity of a given module. When the complexity exceeds this value, it becomes very difficult to prove, understand and modify the module. However, in some circumstances, it may be appropriate to relax the restriction and permit modules with a complexity as high as 15.

Analyzing the cyclomatic complexity metric measured one can observe the low risk of each function and all tools measured it in the same way.

In relation to the MISRA-C rules, as each tool verifies a subset of the rules defined in this standard, the results are different. However, the violations relationed with rules that are included in both RMS and Understand tool have been detected by both tools.

The accepted values for the metrics are defined based on the specific project requirements, project quality criteria or sector best practices. Depending on the metrics required for a project, one or more tools can be used. By this reason a selection of some MISRA-C and other standard to be applied shall be done and each specific violation and quality notice shall be analysed to check the suitability of applied the rule or not.

Taking into account all the obtained results, we can concluded that:

- the functions and file have a good maintanability due to the maintainability Index is >85 in both of them
- the functions have little logic and low risk regarding to the cyclomatic complexity values
- the functions and file have an appropiate size and inside the recommendations due to line metrics and Halstead metrics.
- there are some misra-c rules violations and quality notice although these shall be taken into account only in case they are related to the selected ruled to be applied.
- there are some functions never used

In addition to these, as each existing static analysis tool implements different and very specific techniques (code metrics analysis, semantic analysis, context analysis -interactions between multiple functions calls-, creation of new rules, support coding rules/standard rules, ...) to achieve the required assessment or verification objectives, it is recommended to select different static analysis to cover all the commom areas where problems can occur.

5 Conclusions

This report presents experiments with various static analysis techniques:

1. formal verification with Frama-C/WP (see Chapters 2 and 3)
2. static analysis methods (see Chapter 4)

Deductive verification with Frama-C/WP allows to prove with mathematical strength that software satisfies it functional specification. This approach has been applied to railway software [11] and other software components[12] before. The technical challenge for the software analyzed in the report at hand is the extensive use of low-level bit operations.

Verification with Frama-C/WP is intended mostly for the level of software components. Thus, one can imagine that this approach could replace tests on the level of software components but surely not for software integration and software/hardware integration.

Moreover, deductive verification with Frama-C/WP works most efficiently if there is already a sufficiently precise *informal* specification. This was not the case for the `Bitwalker` which only was accompanied with a high-level description of its intended functionality. However, neither the admissible inputs nor the expected results (including error conditions) were properly described. Instead, this information had to be discovered from the implementation and cross checked with the software designer.

Since formal verification is a very precise but also relatively expensive technique, it does not make sense to use it without using cheaper verification techniques first. Specifically, we highly recommend simpler static analyses from Chapter 4 to quickly identify code deficiencies and to fix them *before* a tool like Frama-C/WP is applied.

References

[1] IEC SC 65A. Functional safety of electrical/electronic/programmable electronic safety-related systems, part 3 software requirements. Technical Report IEC 61508, The International Electrotechnical Commission, 2010.

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

[3] ANSI/ISO C Specification Language. `http://frama-c.com/acsl.html`, March 2014.

[4] Frama-C Software Analyzers. `http://frama-c.com`, March 2014.

[5] WP Plug-in. `http://frama-c.com/wp.html`, March 2014.

[6] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Homepage of the Alt-Ergo Theorem Prover. `http://alt-ergo.lri.fr/`, 2013.

[7] Clark Barrett and Cesare Tinelli. Homepage of CVC4. `http://cvc4.cs.nyu.edu/web`, 2014.

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

[9] ISO. ISO C Standard 1999. Technical report, ISO/IEC JTC 1, 1999. ISO/IEC 9899:1999 draft.

[10] Loïc Correnson, Pascal Cuoq, Florent Kirchner, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual*. `http://frama-c.com/download/user-manual-Neon-20140301.pdf`.

[11] Virgile Prevosto, Jochen Burghardt, Jens Gerlach, Kerstin Hartig, Hans Werner Pohl, and Kim Voellinger. Formal specification and automated verification of railway software with frama-c. In *INDIN*, pages 710–715, 2013.

[12] Kim Völlinger. Einsatz des Beweisassistenten Coq zur deduktiven Programmverifikation. Master's thesis (Diplomarbeit), Humboldt-Universität zu Berlin, August 2013.

[13] J.; Oman, P.W.; Hagemeister and D. Ash. A Definition and Taxonomy for Software Maintainability, 1991.

[14] Bruce Lowther Dan Coleman, Dan Ash and Paul Oman. Using metrics to evaluate software system maintainability. *IEEE Computer*, 27(8):44–49, 1994.