OETCS/WP4/D4.2.2

openETCS

ITEA2
INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

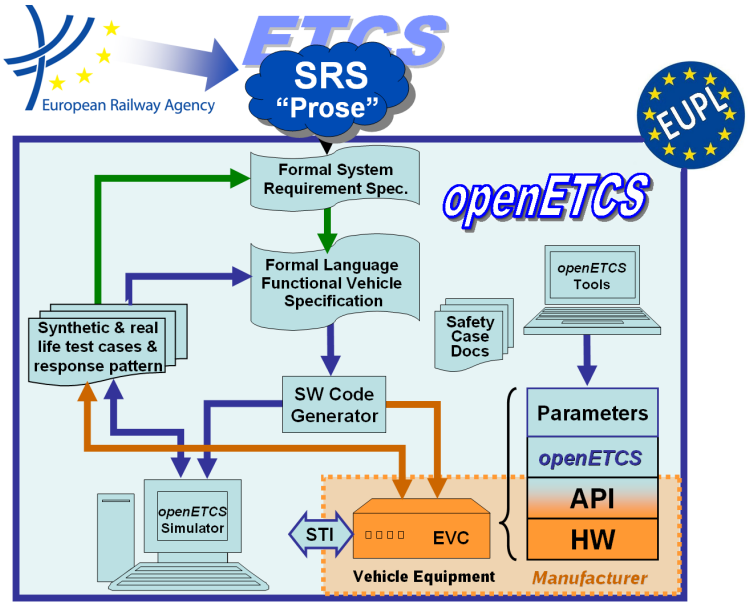
ITEA2 Project
Call 6 11025
2012 – 2015

Work Package 4: “Validation & Verification Strategy”

Preliminary Validation and Verification Report on Implementation/Code

Marc Behrens, Jens Gerlach, Kim Völlinger, Andreas Carben and Izaskun de la Torre

April 2014



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**
April 2014

Preliminary Validation and Verification Report on Implementation/Code

Marc Behrens

DLR, WP4 Leader

Jens Gerlach, Kim Völlinger, Andreas Carben

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

While major parts of the functionality of Subset 026 are developed 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 C-code developed in the OpenETCS project. 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 (Germany). Figure 1.1 gives an overview on the software that is in the focus of this report.

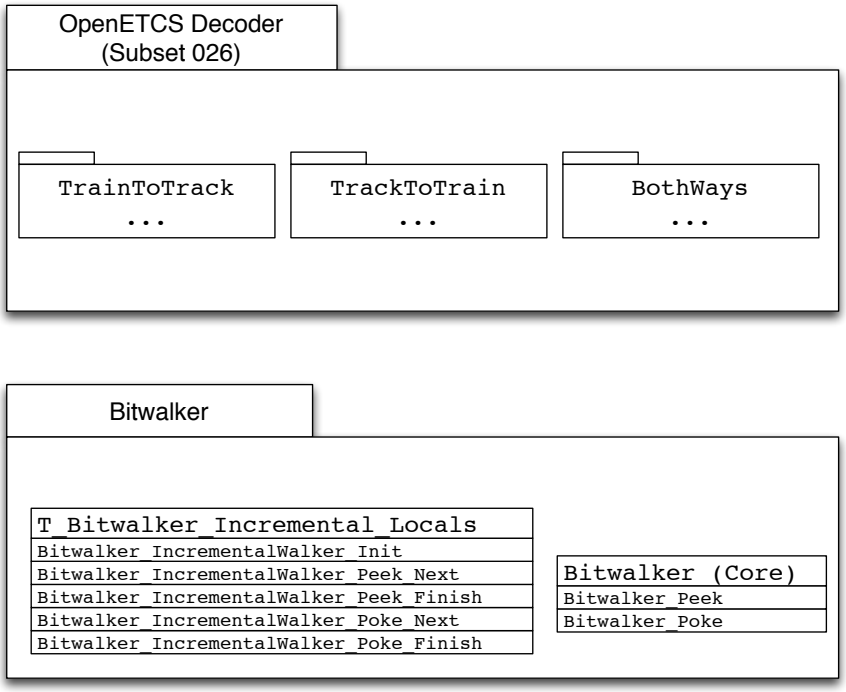


Figure 1.1. The place of `Bitwalker` with the OpenETCS software

The OpenETCS decoder is a large collection of functions dedicated to the reading of 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 the other way round, 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). The Frama-C tool, which is developed by the French project partner CEA LIST, has been used for some of the analyses.

The ultimate verification goals 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, report, we only 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

Section 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 Section 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. using the Frama-C verification platform to establish a formal proof that these C functions do not raise runtime errors when called according to their formal specification.

In Section 4 we report about the results of a broad range of static analyses. These methods are aimed at finding well-known quality deficiencies that might occur in C or C++ software.

Thus, 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 Section 5 we draw preliminary conclusions 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 and explain various aspects of ACSL.

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, however, 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.

```
#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);

  printf("\n");
}
```

Listing 2.3. Some simple test cases for `abs_int`

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.

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 funtion. 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.

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

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 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 use 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.

```
#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`

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
```

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:[2]

  > To show that all software modules, elements and subsystems interact correctly to perform their intended function and do not perform unintended functions

  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

[2] See IEC 61508-3 Table 1

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

FiXme Fatal: improve this intro

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

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

We introduce a method to achieve these goals in section 3.1. Moreover, it is our intention to elaborate the method and in particular the associated tools.

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

Finally, we give an overview about the still open issues in section 3.7.

3.1 Verification Method

In this section we introduce our method of choice along with the used tools. We use a deductive verification approach to formally prove that a function fulfills its specification. The foundations for deductive verification are axiomatic semantics as formulated by Hoare [**?**]. Figure 3.1 shows the method with the involved verification tools.

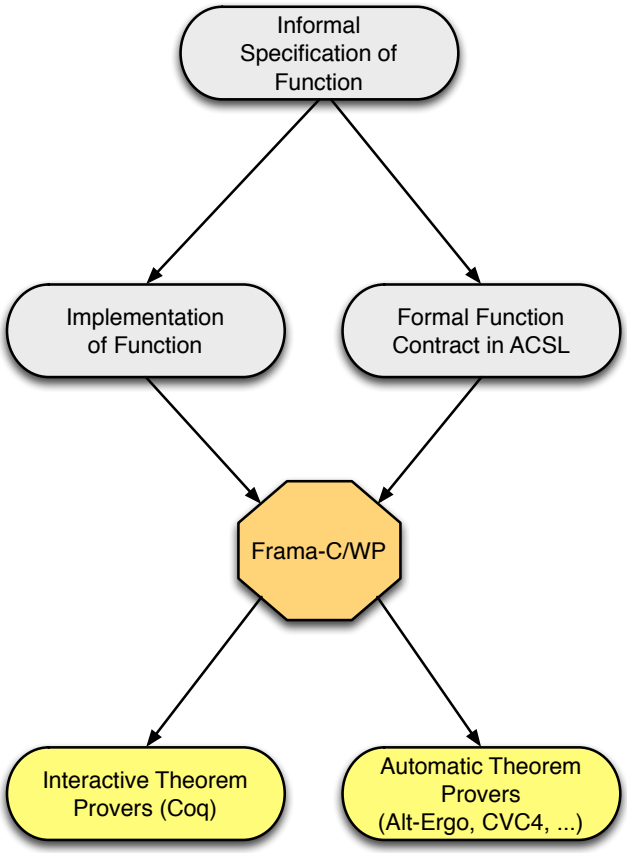


Figure 3.1. Deductive verification of C code with Frama-C/WP.

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

Moreover, it is the specification language associated with the verification platform Frama-C [**?**] which we use along with its plug-in Frama-C/WP [**?**]. 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 the implementation of the function satisfies its contract.

Figure 3.1 shows that we apply the automatic theorem provers Alt-Ergo [**?**] and CVC4 [**?**] and then, if necessary, apply the interactive theorem prover Coq [**?**] for the still unproven conditions. Moreover, unproven conditions motivate to give some extra information in the form of axioms,

lemmas and assertions in ACSL, since they can ease the search of a proof. One need to be careful with axioms because they can yield contradictions and thus make the proof system unsound.

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

The verifiers 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 6, not possible to write meaningful test cases; let alone to formally verify the functionality of the bitwalker functions.

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

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. In fact, this happened in a first version.
- There could also be an error in the implementation which would then be present also in the specification, thus leading to the claim “the code works as implemented”.

In order to avoid these problems we submitted our informal specification for review by the domain experts.

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 original provided. Of course, a specification derived from the source code by the verifier must be subject to a review of the domain experts.

At his 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`.
- 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"[3] 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 looking at the code.

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

Figure 3.2 shows a normalized representation of `Bitwalker_Peek` that is enhanced with static ACSL assertions. 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`

These potential runtime errors 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 returned 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 faces the problem that it

- 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`

3.3 Informal Specifications

Before we provide an informal specification of `Bitwalker_Peek` and `Bitwalker_Poke`, respectively, we introduce some auxiliary concepts and formulate general assumptions. We would also like to point out the following: When we speak of *integers*, then we refer to the infinite set of mathematical integers $\{\ldots, -1, 0, 1, \ldots\}$ and not to one of the many finite representation provided by the type system of C. This distinction is important because mathematical integers usually play an important role in ACSL specifications.

3.3.1 Basic Concepts

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

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

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

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

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

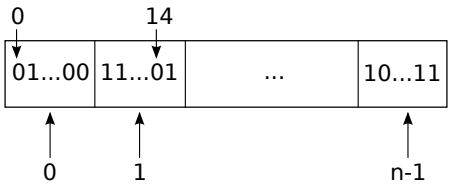


Figure 3.4. Array indices and bit indices in a bit stream

- 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 $k = i \pmod 8$ within this byte. Note that in Figure 3.4 bytes and bits are indexed in increasing order from the *left*. On the byte level, however, bits are often indexed from the *right*. For example, to access the k-th bit of a byte `a` one can shift this bit to the right by $7 - k$ and extracts then the now rightmost bit by performing a bit-wise *and* with the value 1

```
(a >> (7-k)) & 1
```

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

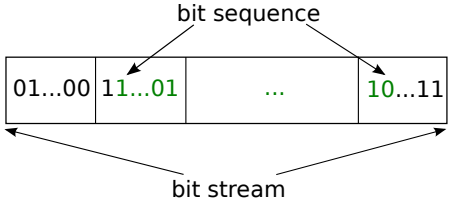


Figure 3.5. A bit sequence within a bit stream

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

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

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

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

Its function signature reads as follows:

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

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. We were wondering why the implementation maps an illegal input to a legitimate 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 starts.
- `Length` is the length of the bit sequence.
- `Bitstream` is the array which provides the bit stream.
- `BitstreamSizeInBytes` is the length of the array containing the bit stream.
- `Value` is the integer which shall be converted into a bit sequence.

Preconditions

The following conditions shall hold for the function arguments:

- `Bitstream` is a valid array of length `BitstreamSizeInBytes`
- `Startposition + Length` is less than `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 to a bit sequence and writes it into a bit stream.

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

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

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 Tests for `Bitwalker_Peek` and `Bitwalker_Poke`

In this section we show some tests for `Bitwalker_Peek` and `Bitwalker_Poke`. These tests were derived from the informal specification in Section 3.3.

We use the C++ class `boost::dynamic_bitset` in order to represent `bit sequences` in our tests. This class[4], which is part of the Boost libraries, provides a higher-level and easier to use interface to bit sequences than is possible in C.

Specifically, we use in our C++ test code 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`

Listings 3.4 and 3.5 show fragments of our test code for `Bitwalker_Peek` and `Bitwalker_Poke`, respectively.

While 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.
- It provided some assurance that our re-implementations of `Bitwalker_Peek` and `Bitwalker_Poke` that we use in Section 3.5 do not behave differently than the original implementations.

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

```
#include "test_bitwalker.h"

void test_peek(unsigned int start,
               unsigned int length,
               Bytestream bytes,
               uint64_t expected_value)
{
    std::stringstream msg;

    if (length >= 64)
    {
        msg << "length = " << length << " must be less than 64";
        throw std::logic_error(msg.str());
    }

    if (start >= UINT_MAX - length)
    {
        msg << "start = " << start << " must be less than " <<
           UINT_MAX - length;
        throw std::logic_error(msg.str());
    }

    const uint64_t value = Bitwalker_Peek(start, length, bytes.data
        (), bytes.size());
    Bitstream original(bytes.rbegin(), bytes.rend());

    if (value != expected_value)
    {
        msg << std::endl;
        msg << "value = " << value << std::endl;
        msg << "does not match" << std::endl;
        msg << "expected_value = " << expected_value << std::endl;
        msg << "start = " << start << std::endl;
        msg << "length = " << length << std::endl;
        msg << "byte array = " << bytes << std::endl;
        throw std::runtime_error(msg.str());
    }

    test_peek_normal_case(original, value, start, length);
}
```

Listing 3.4. Test code for `Bitwalker_Peek`

```
#include "test_bitwalker.h"

void test_poke(unsigned int start,
               unsigned int length,
               Bytestream bytes,
               uint64_t value,
               int expected_code)
{
    std::stringstream msg;

    if (length >= 64)
    {
        msg << "length = " << length << " must be less than 64";
        throw std::logic_error(msg.str());
    }

    if (start >= UINT_MAX - length)
    {
        msg << "start = " << start << " must be less than " <<
           UINT_MAX - length;
        throw std::logic_error(msg.str());
    }

    const Bitstream original(bytes.rbegin(), bytes.rend());
    const int exit_code = Bitwalker_Poke(start, length, bytes.data()
       , bytes.size(), value);

    if (exit_code != expected_code)
    {
        msg << std::endl;
        msg << "exit_code = " << exit_code << std::endl;
        msg << "does not match" << std::endl;
        msg << "expected_code = " << expected_code << std::endl;
        msg << "start = " << start << std::endl;
        msg << "length = " << length << std::endl;
        msg << "value = " << value << std::endl;
        msg << "byte array = " << bytes << std::endl;
        throw std::runtime_error(msg.str());
    }

    const Bitstream changed(bytes.rbegin(), bytes.rend());
    test_poke_normal_case(original, changed, exit_code, value, start
       , length);
}
```

Listing 3.5. Test code for `Bitwalker_Poke`

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 the contracts 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 3.6 shows an ACSL contract with the main operation modes of `Bitwalker_Peek`. Note that we have labeled various properties of the contract. 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) - 1;

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

Listing 3.6. Formal specification of `Bitwalker_Peek` in ACSL

The structure of this contract is as follows:

Default behavior

- The property `readable_bitstream` use the built-in ACSL predicate `\valid_read`. This expresses that 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 less 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`, respectively.
  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
  ```

  The main difference is that we reformulate the division inherent in the shift operation as a multiplication. Moreover, 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 no 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, which must be based on Formula (2) on Page 25, will be part of a later release 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 Frama-C/WP.

3.5.2 Code Annotations for `Bitwalker_Peek`

Listing 3.7 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).

```
#include "Bitwalker_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; // error: invalid_bit_sequence

  //@ 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 shift = Bitstream[byte_index] >> bit_index;
    unsigned int bit_as_byte = shift & 1;
    //@ assert bit_as_byte == 0 || bit_as_byte == 1;

    retval = 2 * retval + bit_as_byte;
  }

  return retval;
}
```

Listing 3.7. Implementation of `Bitwalker_Peek` with ACSL loop invariants

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 3.7.

- 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`.

3.5.3 Formal Specification of `Bitwalker_Poke`

Listing 3.8 shows an ACSL contract with the main operation modes of `Bitwalker_Poke`. Again we have labeled some properties of the contract and use for some variables shorter names than in 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[Start/8..(Start + Length)/8];

  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[Start/8..(Start + Length)/8];

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

Listing 3.8. 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 that all addresses starting at `Bitstream` and with offsets in the range `0..BitstreamSize-1` can be safely dereferenced for *reading and writing*.
- The property `valid_length` expresses the requirement that only bit sequences with a length less 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`, respectively.
- 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 where 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 request that no external memory locations are changed when this behavior is active.

Behavior for values that do not fit into the bit sequence

The behavior `value_too_big` describes the case where 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_Peek`. 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 3.9 shows our modified version of `Bitwalker_Poke`.

```
#include "Bitwalker_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[Start/8..(Start + Length)/8];
    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 3.9. 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 simply test that it is greater or equal than $2^{\texttt{Length}}$.

3.6 Formal Verification with Frama-C/WP

FiXme Fatal: not ready for review

In this section we present the current state of the verification results for `Bitwalker_Peek`. Table 3.1 discriminates the results for three different types of verification conditions (VCs).

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

Table 3.1. Verification Results of `Bitwalker_Peek`

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

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

The verification rate for the `rte`-assertions are very low due to the difficulty for Frama-C to deal with bit operations. In order to increase this rate, we will verify the absence of run time errors separately and will provide additional lemmas and axioms to ease the verification. We point out some of the related challenges in section 3.7.

In this section we present the current state of verification results of for `Bitwalker_Poke`. The results are shown in Table 3.2. We listed the different verification conditions row by row like we did for `Bitwalker_Peek`.

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

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

Table 3.2. Verification Results of `Bitwalker_Poke`

3.7 Open Issues

FiXme Fatal: not ready for review

We have seen in this section that WP currently does not deal very well with bit operations. This is due to the fact that WP's memory models do not provide much information about bit operations. As a consequence, the provers have few options to manipulate the proof goal. This problem is known and CEA LIST is working on a solution for the next release of WP.

As a workaround one could introduce axioms which provide additional facts about bit operations. The problem with using axioms is that one can easily introduce wrong facts which lead to contradictions making the whole proof system unsound. Thus, this approach requires a careful review of the added axioms.

Moreover, the chosen automatic theorem provers are generally not very good when it comes to mixing arithmetic and bit operations. There is, however, an automatic theorem prover, namely Z3, which can handle arithmetic and bit operations, using a specific syntax. Frama-C's interface for Z3 does not currently takes advantage of this, but this may change in a future release. We therefore expect a better automatic verification rate for the verification of BitWalker.

Another approach to deal with unproven verification conditions consists in applying an *interactive theorem prover* such as Coq. Using Coq's rich support for proof manipulation would certainly be very helpful for the discharge of more proof obligations.

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, 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. 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. 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.

Five 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.

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

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.

A simplified formula to calcule the cyclomatic complexity is: V(g) = P + 1, where P are the predicate nodes.

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

Furthermore, 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.
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..

Table 4.1. Quality Notices

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 non-deterministic. 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.
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.

Table 4.1. Quality Notices

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.
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_*

RSM also allows to customize the desired output providing standard metrics and a combination of features.

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

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

As mentioned previously CENELEC Standard requires the use of a package of techniques. With the use of the RSM tool the Limited Size and Complexity in Functions, Subroutines and Methods high recomended technique and the Coding Standard mandatory technique are covered. At this point the fulfillment of some of the MISRA-C Standard rules has been checked.

At following we provide a summary of the obtained results.

The table below indicates the total quality profile (Summary by notice type) for the bitwalker code which 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
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_*

At following some code metrics by file will be shown, but previously an explanation is included. 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.

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.

Table 4.4. File Summary

Metrics	Bitwalker.h	Bitwalker.c	main.c	opnETCS.h	opnETCS _Decoder.h
LOC[5]	15	58	45	884	62

[5] Lines of Code

Table 4.4. File Summary

Metrics	Bitwalker.h	Bitwalker.c	main.c	opnETCS.h	opnETCS _Decoder.h
eLOC[6]	15	40	40	823	62
lLOC[7]	11	28	23	760	61
Comment	16	29	61	822	15
Lines	41	109	127	1249	84

The table below provides information regarding standard functional metrics such as cyclomatic complexity and others.

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.

As it was shown previously the cyclomatic complexity metric measures the complexity of the code and it is calculated as V(g) = P + 1, where P are the predicate nodes. The result obtained in the calculation of the cyclomatic complexity defines the number of independent paths within a piece of code and determines the upper bound on the number of tests that must be performed to ensure that each statement is executed at least once.

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.

Table 4.5. Functional Summary

Metrics	Bitwalker.c	main.c
File Function Count	7	1
Total Function LOC	49	40
Total Function eLOC	31	35
Total Function lLOC	27	23
Total Function Params	20	0
Total Cyclo Complexity	13	1
Total Function Pts LOC	0.5	0.4
Total Function Pts eLOC	0.3	0.3
Total Function Pts lLOC	0.2	0.2
Total Function Return	10	1
Total Function Complex	43	2

[6]Effective Lines of Codes

[7]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. Functional Summary

Metrics	Bitwalker.c	main.c
Max Function LOC	16	40
Max Function eLOC	12	35
Max Function lLOC	9	23
Average Function LOC	7.00	40
Average Function eLOC	4.43	35
Average Function lLOC	3.86	23
Max Function Parameters	5	0
Max Function Returns	3	1
Max Interface Complex	8	1
Max Cyclomatic Complex	5	1
Max Total Complexity	13	2
Avg Function Parameters	2.86	0.00
Avg Function Returns	1.43	1.00
Avg Interface Complex	4.29	1.00
Avg Cyclomatic Complex	1.86	1.00
Avg Total Complexity	6.14	2.00

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

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.

Table 4.6. 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

Table 4.6. Function Metrics

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
main			
Param: 0	Return: 1	Cyclo Vg: 1	Comment: 47
LOC: 40	eLOC: 35	lLOC: 23	Lines: 101

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

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

```
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

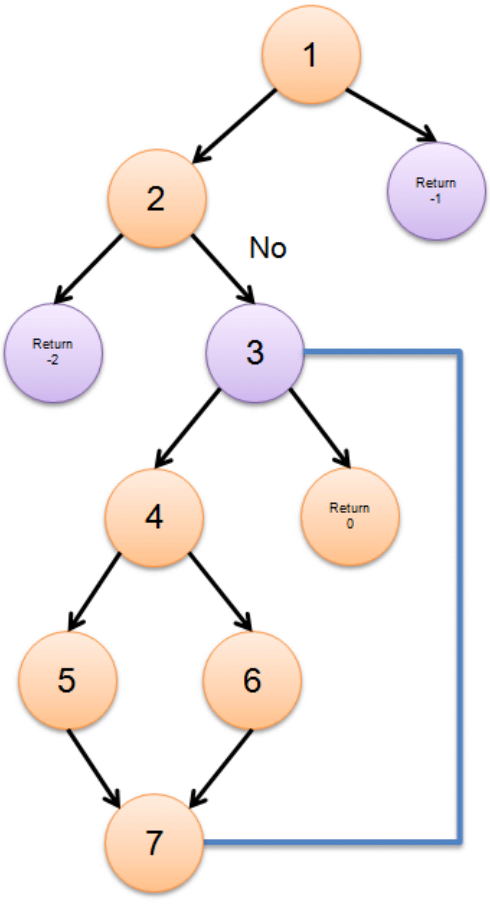


Figure 4.1. Bitwalker_Poke Flow

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 4.7. LocMetrics Tool Results

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

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 MISRA-C rules by partners it has been decided that some of them are not to be implemented/approved due to its application can get worse understanbility of the code.

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

Table 4.8. Status of MISRA Rules

MISRA Rule	Status
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 4.9. 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 4.10. Function Complexity metrics

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

Table 4.10. Function Complexity metrics

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
Strict Cyclomatic:	1
Essential:	1
Max Nesting:	0
Count Path:	1
main	
Cyclomatic:	1
Modified Cyclomatic:	1
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 4.11. File Metrics

Metrics	Bitwalker.h	Bitwalker.c	main.c	opnETCS.h	opnETCS _Decoder.h
Lines:	41	109	127	1249	84
Comment Lines:	20	33	65	822	20
Blank Lines:	7	23	22	180	2
Preprocessor Lines:	4	1	4	1	1
Code Lines:	11	57	41	883	61
Inactive Lines:	0	0	0	0	0
Executable Code Lines:	0	30	33	0	0
Declarative Code Lines:	11	15	35	822	61
Execution Statements:	0	28	12	0	0
Declaration Statements:	11	15	12	760	61
Ratio Comment/Code:	1.82	0.58	1.59	0.93	0.33
Units	0	7	1	0	0

Table 4.12. 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

Table 4.12. Function code Metrics

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
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

Table 4.12. Function code Metrics

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
Execution Statements:	12
Declaration Statements:	2
Ratio Comment/Code:	0.35
main	
Lines:	102
Comment Lines:	47
Blank Lines:	19
Code Lines:	41
Inactive Lines:	0
Executable Code Lines:	33
Declarative Code Lines:	25
Execution Statements:	12
Declaration Statements:	11

Table 4.12. Function code Metrics

Ratio Comment/Code:	1.15

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 4.13. 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 4.14. Uninitialized Items

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

Table 4.15. 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 4.16. 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).
core.CallAndMessage	Check for logical errors for function calls and Objective-C message expressions (e.g., uninitialized arguments, null function pointers).

Table 4.16. Aspects checked

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:

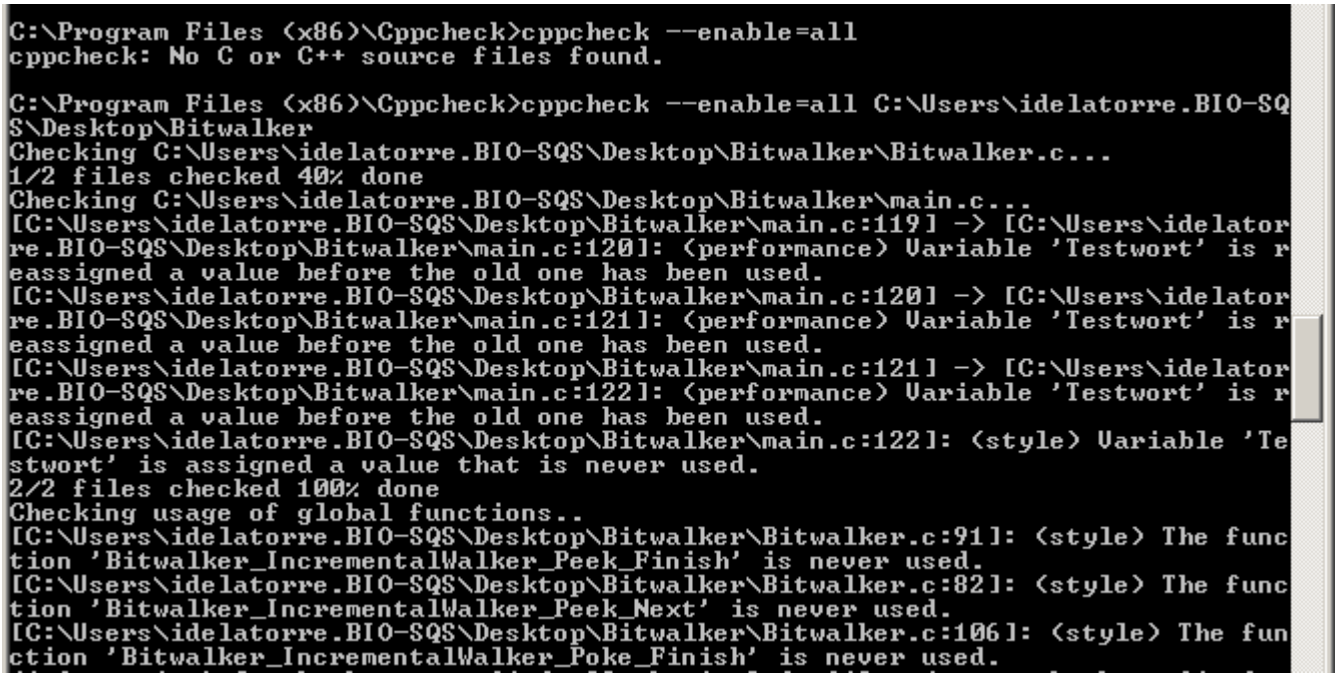


Figure 4.4. cppcheck results

4.7 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.

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

For example, the table below shows that there is a minimun deviation regarding to the cyclomatic complexity obtained with the RSM or Understand and LocMetrics tools

Table 4.17. File Cyclomatic Complexity comparation

File	RSM	Understand	LocMetrics
Bitwalker.c	13	13	15
main.c	1	1	1

Table 4.18. function Cyclomatic Complexity comparation

Function	RSM	Understand
Bitwalker_Peek	3	3
Bitwalker_Poke	5	5
Bitwalker_IncrementalWalker_Init	1	1
Bitwalker_IncrementalWalker_Peek_Next	1	1
Bitwalker_IncrementalWalker_Peek_Finish	1	1
Bitwalker_IncrementalWalker_Poke_Next	1	1
Bitwalker_IncrementalWalker_Poke_Finish	1	1
main.c	1	1

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

- the complexity of bitwalker.c file is 13 that exceeds the 10 value so the bitwalker code has moderate risk. Thus, we might split it into smaller modules.
- there are some misra-c rules violations and quality notice. It would be recommendable to modify the specific lines if it is possible in order to improve code quality.

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