OETCS/WP4/D4.2.2

openETCS

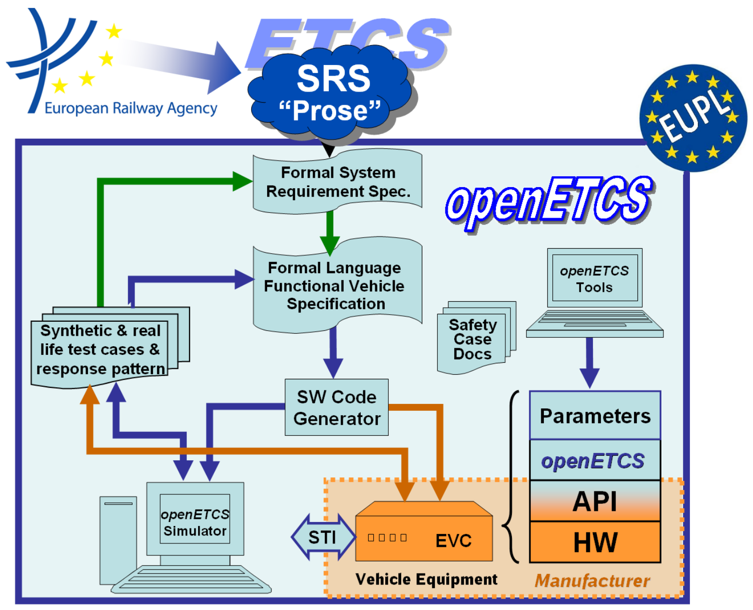
ITEA2 Project
Call 6 11025
2012 – 2015

Work Package 4: “Validation & Verification Strategy”

First Validation and Verification Report on Implementation/Code

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

December 2013



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**
December 2013

First Validation and Verification Report on Implementation/Code

Marc Behrens
DLR, WP4 Leader

Jens Gerlach, Kim Völlinger, Andreas Carben
Fraunhofer FOKUS, WP4.3 Task Leader (Validation and Verification of Implementation/Code)

Izaskun de la Torre
Software Quality Systems S.A.

Description of work

Prepared for openETCS@ITEA2 Project

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

Table of Contents

Figures and Tables

Figures

Tables

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).

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.

In Section 2 we report about the results of a broad range of static analyses. These methods are aimed at finding well-known deficiencies that might occur in C or C++ software.

In Section 3 we take a more formal approach by

1. formally specifying 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 the C code satisfies the formal specification.

Regarding the more formal approach it must be pointed out that 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.

2 SQS

2.1 Introduction

In this section 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 some subset of rules defined in the MISRA C Standard.

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, are a precise measure of the code complexity, and the more complex the code is, more likely it will contain masked bugs.

Five different static analysis tools have been used during the code verification activities in order to assess the quality of the results and ensure code quality.

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

2.2 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.

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

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

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 + 2, where E are the edges and N are the nodes.

- 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 2.1. 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.

Table 2.1. Quality Notices

Quality Notice No. 11 Emit a quality notice when open and closed brackets '[ ]' are not balance within a file. Rationale: This type of error is always caught by the compiler as a syntax error. However, a compiler can be told to ignore source code by using preprocessor directives like #if ... #endif. This is a way to "comment" out large blocks of code. However, the code still looks like operational code to the maintainer as it is not a comment. Many hours can be wasted working on dead code. This quality notice serves to warn you of this dead code that should be removed or converted to actual comment form.	**Quality Notice No. 12** Emit a quality notice when open and closed parenthesis '( )' are not balance within a file. Rationale: This type of error is always caught by the compiler as a syntax error. However, a compiler can be told to ignore source code by using preprocessor directives like #if ... #endif. This is a way to "comment" out large blocks of code. However, the code still looks like operational code to the maintainer as it is not a comment. Many hours can be wasted working on dead code. This quality notice serves to warn you of this dead code that should be removed or converted to actual comment form..
Quality Notice No. 13 Emit a quality notice when a 'switch' statement does not have a 'default' condition. Rationale: A "switch" statement must always have a default condition or this logic construct is 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.

Table 2.1. Quality Notices

Quality Notice No. 23 Emit a quality notice when the '?' or the implied if-then-else construct has been identified. Rationale: The ? operator creates the code equivalent of an "if" then "else" construct. However the resultant source is far less readable.	**Quality Notice No. 24** Emit a quality notice when an ANSI C++ keyword is identified within a *.c or a *.h file. Rationale: In C source code it is possible to find variable names like "class". This word is a key word in C++ and would prevent this C code from being ported to the C++ language.
Quality Notice No. 25 (Deprecated RSM 6.70) When analyzing *.h files for C++ keywords, assume that *.h can be both C and C++. Rationale: A *.h file can be either a C or C++ source file. If a *.h file is assumed to be from either language, then RSM will not emit C keyword notices in *.h file, only for *.c files.	**Quality Notice No. 26** Emit a quality notice when a void * is identified within a source file. Rationale: A "void *" is a type-less pointer. ANSI C and C++ strives to be type strict. In C++ a "void *" breaks the type strict nature of the language which can give rise to anomalous run-time defects.
Quality Notice No. 27 Emit a quality notice when the number of function return points is greater than the specified maximum. Rationale: A well constructed function has one entry point and one exit point. Functions with multiple return points are difficult to debug and maintain.	**Quality Notice No. 28** Emit a quality notice when the cyclomatic complexity of a function exceeds the specified maximum. Rationale: Cyclomatic complexity is an indicator for the number of logical branches within a function. A high degree of V(g), greater than 10 or 20, indicates that the function could be broken down into a more modular design of smaller functions.
Quality Notice No. 29 Emit a quality notice when the number of function input parameters exceeds the specified maximum. Rationale: A high number of input parameters to a function indicates poor modular design. Data should be grouped into representative data types. Functions should be specific to one purpose.	**Quality Notice No. 30** Emit a quality notice when a TAB character is identified within the source code. Indentation with TAB will create editor and device dependent formatting. Rationale: Tab characters within source code create documents that are print and display device dependent. The document may look correct on the screen but it may become unreadable when printed.
Quality Notice No. 31 Emit a quality notice when class comment percentage is less than the specified minimum. Rationale: A programmer must supply sufficient comments to enable the understandability of the source code. Typically a comment percentage less than 10 percent is considered insufficient.	**Quality Notice No. 43** Emit a quality notice when the key word 'continue' has been identified within the source code. Rationale: The use of 'continue' in logical structures causes a disruption in the linear flow of the logic. This style of programming can make maintenance and readability difficult.
Quality Notice No. 46 Emit a quality notice when function, struct, class or interface blank line percentages are less than the specified minimum Rationale: The amount of blank lines in a file can indicate the degree of readability in the file. It indicates the author intended his work to be human consumable.	**Quality Notice No. 47** Emit a quality notice when the file blank line percentage is less than the specified minimum Rationale: The amount of blank lines in a file can indicate the degree of readability in the file. It indicates the author indented his work to be human consumable.
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.

Table 2.1. Quality Notices

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 2.2. 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

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 2.3. 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_*

The following table shows some code metrics by file.

Table 2.4. File Summary

Metrics	Bitwalker.h	Bitwalker.c	main.c	opnETCS.h	opnETCS _Decoder.h
LOC[1].	15	58	45	884	62
eLOC[2]	15	40	40	823	62
lLOC[3]	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.

[1]Lines of Code
[2]Effective Lines of Code
[3]Logical Statements Lines of Code

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 2.5. 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
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

Table 2.5. Functional Summary

Metrics	Bitwalker.c	main.c
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.

Table 2.6. 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
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

Table 2.6. Function Metrics

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 .

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

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

    if (MaxValue < Value)
        return -2;

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

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

Listing 2.1. Bitwalker_Poke

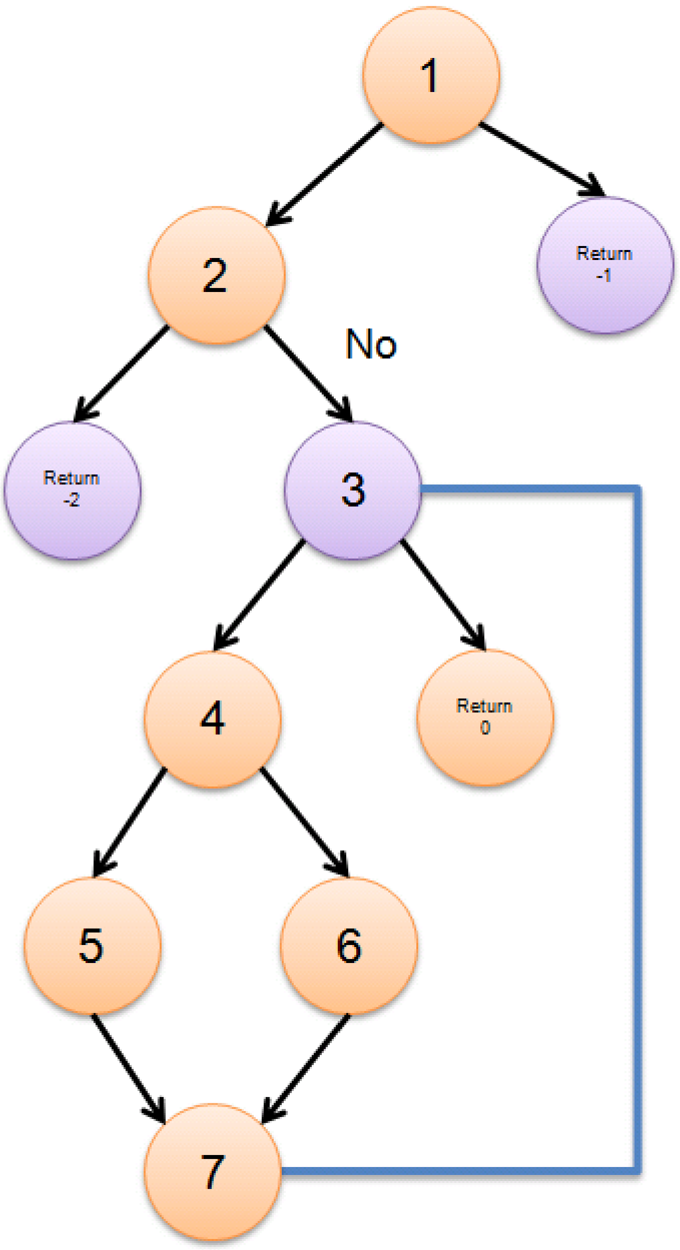


Figure 2.1. Bitwalker_Poke Flow

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.

2.3 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 2.7. 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

2.4 Understand tool Results

[Understand] is a cross-platform, multi-language, maintenance-oriented IDE (Interactive Development Environment). It is designed to help maintain and understand large amounts of legacy or newly created source code. With this tool SQS has checked MISRA-C:2004 and code metrics (lines of code, complexity, object cross reference, invocation tree, Unused Items and others)

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.

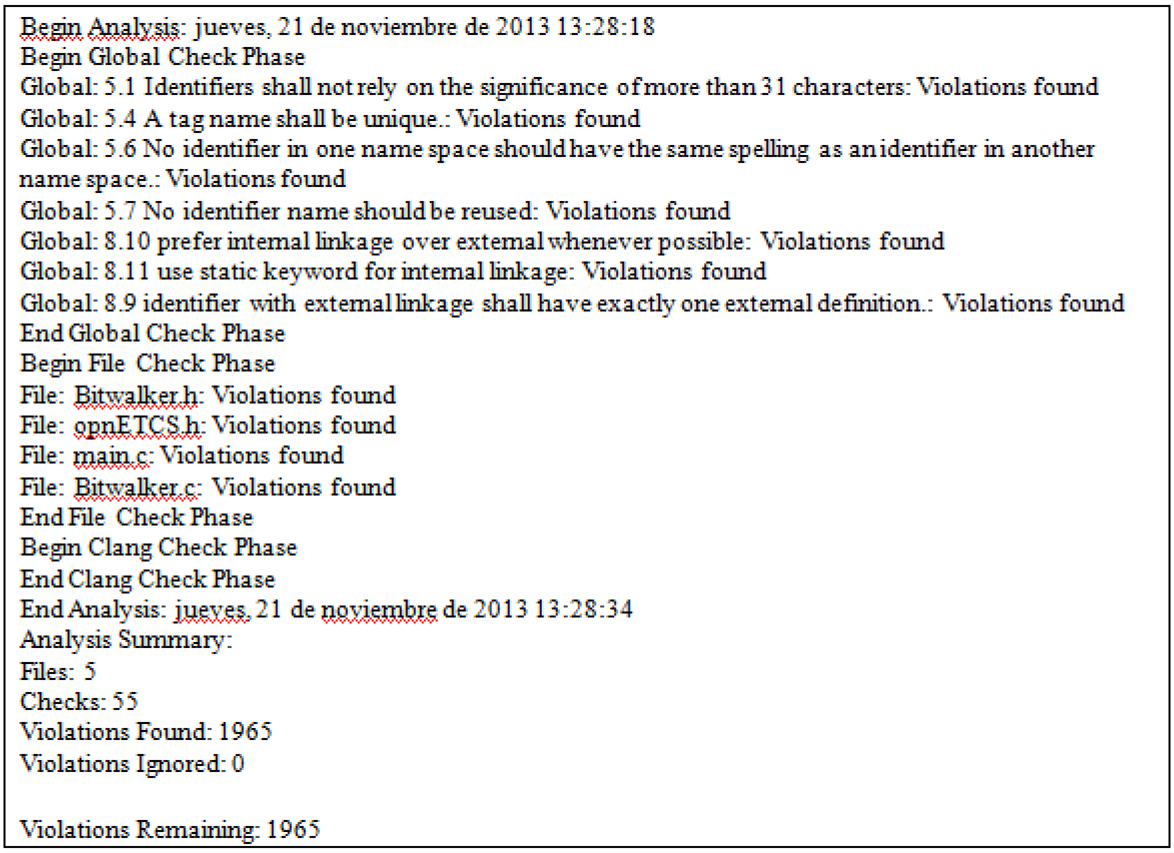
- **Structures and unions**
  - 18.4 (req): Unions shall not be used.

- **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.

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 2.2. MISRA-C Rules results

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

Table 2.8. 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

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.

Table 2.9. Function Complexity metrics

Bitwalker_Peek

Table 2.9. Function Complexity metrics

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
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

Table 2.9. Function Complexity metrics

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

Table 2.10. 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 2.11. Function code Metrics

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

Table 2.11. Function code Metrics

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

Table 2.11. Function code Metrics

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
Ratio Comment/Code:	1.15

2.5 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:

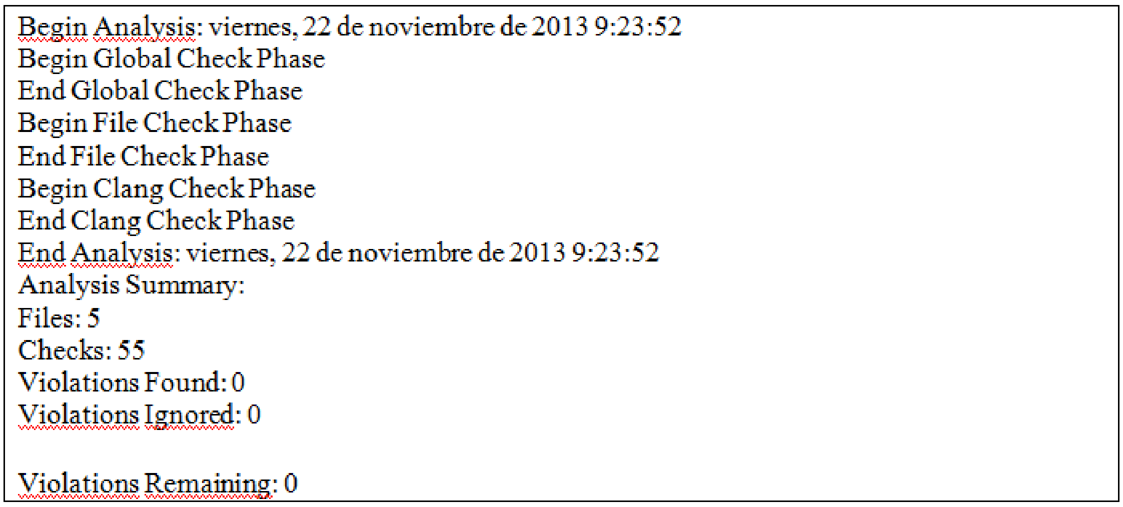
Table 2.12. Aspects checked

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).
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.

Table 2.12. Aspects checked

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.

After run this analysis no violation has been found.



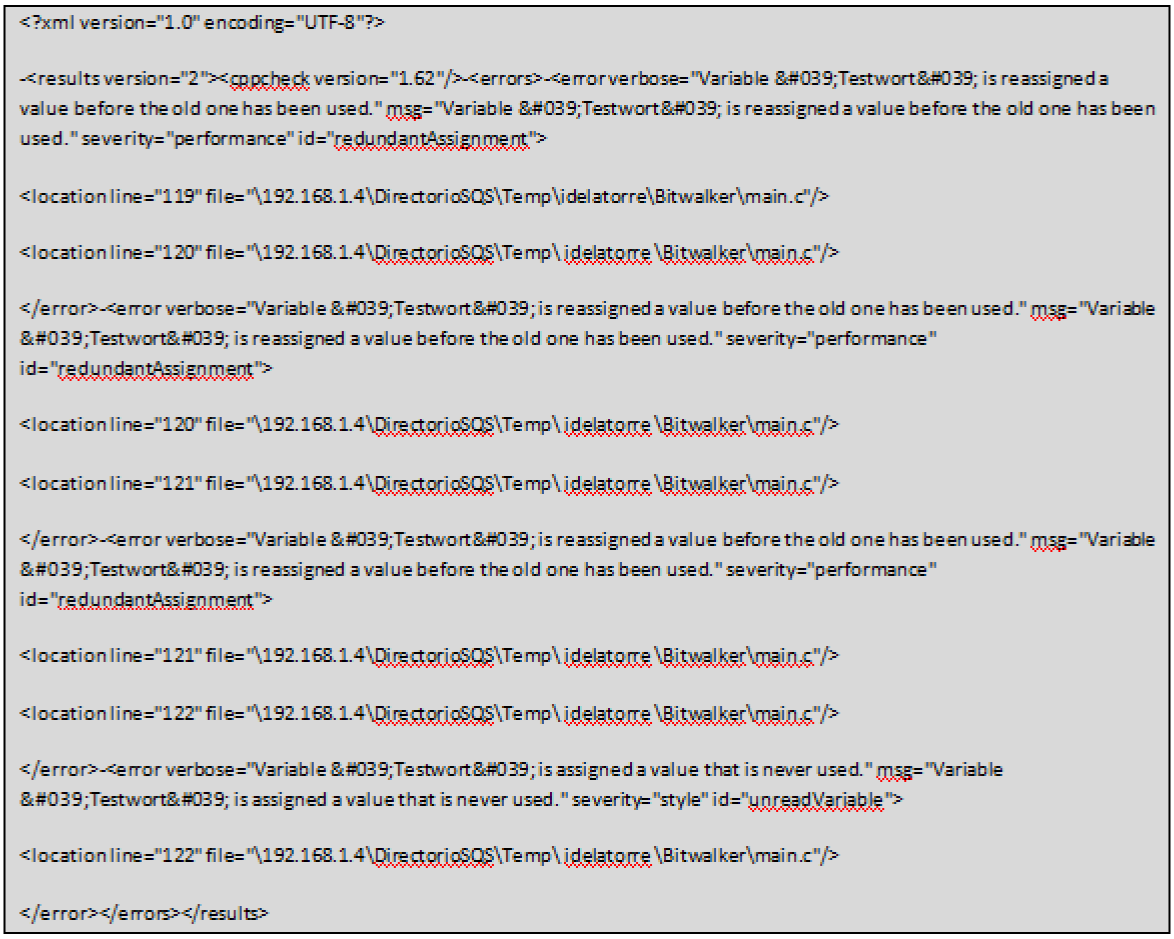
```
Begin Analysis: viernes, 22 de noviembre de 2013 9:23:52
Begin Global Check Phase
End Global Check Phase
Begin File Check Phase
End File Check Phase
Begin Clang Check Phase
End Clang Check Phase
End Analysis: viernes, 22 de noviembre de 2013 9:23:52
Analysis Summary:
Files: 5
Checks: 55
Violations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

Violations Remaining: 0
```

Figure 2.3. Clang Analysis results

2.6 CPPcheck tool Results

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

The results of the tool are the following:



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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2.4. cppcheck results

2.7 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 2.13. File Cyclomatic Complexity comparation

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

Table 2.14. 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.

3 Formal Verification of Bitwalker

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

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

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

Subsequently, we use the method for `Bitwalker_Peek` and `Bitwalker_Poke` in section 3.2 and 3.3, respectively. We provide an informal specification, an implementation and a formal specification for each function and present what could have been verified for the implementation.

We discuss the interaction of these functions in section 3.4 where we show how the interaction can be formally specified and present the verification results. Finally, we give an overview about the still open issues in section 3.5.

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.

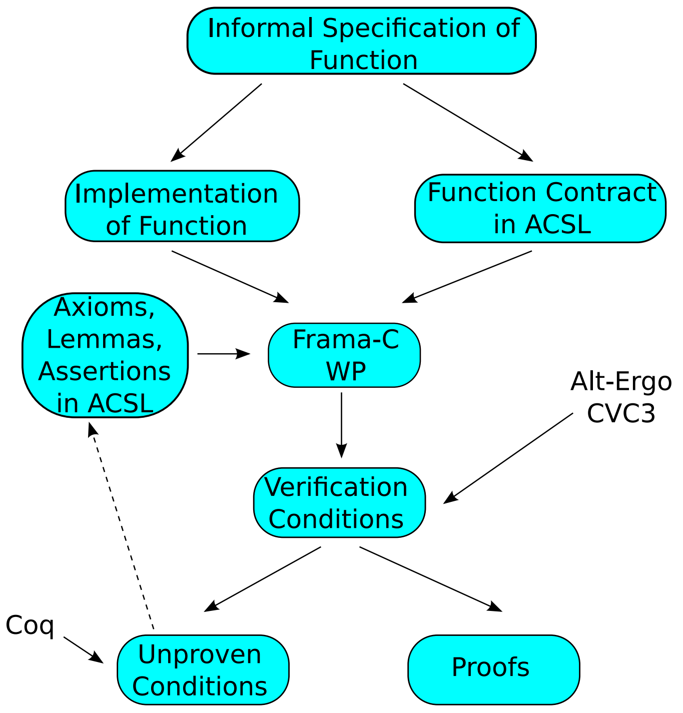

Informal Specification of Function
Implementation of Function
Function Contract in ACSL
Axioms, Lemmas, Assertions in ACSL
Frama-C WP
Alt-Ergo CVC3
Verification Conditions
Coq
Unproven Conditions
Proofs

Figure 3.1. The method to formally verify the Bitwalker.

Starting point is an informal specification of a function with which in mind a implementation is written and on which basis the formal specification is created. The formal specification of a function is a so-called function contract which contains preconditions to express what a function expects from its caller and postconditions to state the guarantees after the execution. The specification language is called ACSL (ANSI/ISO-C Specification Language) [?] 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 WP [?]. Within Frama-C, WP enables the deductive verification of C programs that have been annotated with ACSL. WP generates verification conditions which are submitted to external automatic or interactive theorem provers. A function is then verified if each verification condition is discharged by at least one prover.

Figure 3.1 shows that we first apply the automatic theorem provers Alt-Ergo [?] and CVC3 [?] and then apply the interactive theorem prover Coq [?] for the still unproven conditions in order to automate as much as possible. Moreover, unproven conditions motivate to give some extra information in the form of axioms, lemmas and assertions in ACSL, since they can ease the search of a proof. One need to be careful with axioms because they can yield contradictions and thus make the proof system unsound. This is different for lemmas and assertions because WP will generate additional verification conditions for them.

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

3.2 The Function `Bitwalker_Peek`

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

3.2.1 Informal Specification

We first introduce some auxiliary concepts and formulate general assumptions:

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

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

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

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

  Figure 3.2 shows the difference between array indices and bit indices in a bit stream. The two bit indices, 0 and 14, mark bit positions in the first and second array element, respectively.

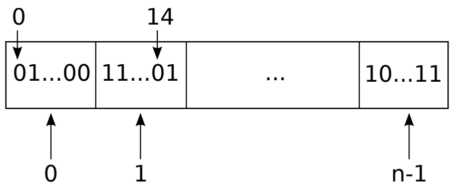


Figure 3.2. Array indices and bit indices in a bit stream

- A *bit sequence* is a consecutive sequence of bits within a bit stream as represented in Figure 3.3.

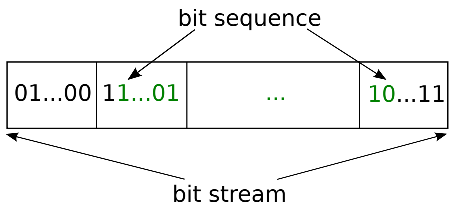


Figure 3.3. A bit sequence within a bit stream

  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

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

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

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

Its function signature reads as follows:

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

The arguments have the following purpose:

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

The following preconditions shall hold for the function arguments:

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

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

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

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

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

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

3.2.2 Implementation

Listing 3.1 shows the C implementation of `Bitwalker_Peek` for which we aim to verify that it fulfills the informal specification. The case where the bit sequence is not valid is handled by the `if`-statement. For a valid sequence the summation of the bits is done in the `for`-loop. The array `BitwalkerBitMaskTable` is a `const` helper array to select a single bit in the `Bitstream`.

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

    uint64_t retval = 0;

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

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

    return retval;
}
```

Listing 3.1. Implementation of `Bitwalker_Peek`

The implementation uses a great amount of bit operations which is quite a challenge for the formal verification. We will discuss this further in section 3.5.

3.2.3 Formal Specification with ACSL

In order to verify that the given implementation of `Bitwalker_Peek` fulfills the informal specification, we have to formalize the specification. Listing 3.2 shows such a formalization in ACSL for `Bitwalker_Peek`.

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

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

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

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

Listing 3.2. Formal specification of `Bitwalker_Peek` in ACSL

We specify a function contract for `Bitwalker_Peek` containing preconditions and postconditions introduced by the key words **`requires`** and **`ensures`**, respectively. In addition, the ACSL language provides the **`assigns`** clause to specify that a function is not allowed to change memory locations other than the ones explicitly listed. When no **`assigns`** clauses are specified, the function is allowed to modify every accessible memory location.

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

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

In both cases we state what the result of `Bitwalker_Peek` shall be as postconditions. In addition, we use a negated form of a predicate called `TooBig` in the last postcondition of the normal case. This postcondition was introduced to verify that the functions `Bitwalker_Peek` and `Bitwalker_Poke` interact correctly. Therefore, we will discuss this postcondition in section 3.4.

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

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

3.2.4 Formal Verification with Frama-C/WP

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 3.5.

3.3 The Function `Bitwalker_Poke`

In this section we examine the function `Bitwalker_Poke` in the same manner as we did it for `Bitwalker_Peek` in section 3.2.

3.3.1 Informal Specification

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

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

The arguments have the following purpose:

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

The following preconditions shall hold for the function arguments:

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

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

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

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

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

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

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

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

3.3.2 Implementation

Listing 3.3 shows the implementation of Bitwalker_Poke which discriminates three cases. The first two are the robustness cases of the informal specification and the last one is the normal case where the function actually writes into the bit stream. Similarly to Bitwalker_Peek a lot of bit operations are used.

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

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

    if (MaxValue < Value)
        return -2;

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

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

Listing 3.3. Implementation of Bitwalker_Poke

3.3.3 Formal Specification with ACSL

Listing 3.4 shows the function contract of `Bitwalker_Poke`. The case independent preconditions of the informal specification are reflected by the first three **`requires`**-clauses at the beginning of the contract. `Bitwalker_Poke` modifies the `Bitstream` and reads the array `BitwalkerBitMaskTable` thus we need to express that the two arrays must have separated memory locations. Therefore, we use the predicate `separated` in the fourth **`requires`**-clause. Furthermore, in the following **`assigns`**-clause we specify the memory locations which can be altered by the function.

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

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

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

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

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

        assigns \nothing;

        ensures \result == -1;

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

        assigns \nothing;

        ensures \result == -2;

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

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

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

    complete behaviors;
    disjoint behaviors;
*/

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

Listing 3.4. Formal Specification of `Bitwalker_Poke`

3.3.4 Formal Verification with Frama-C/WP

In this section we present the current state of verification results of for `Bitwalker_Poke`. The results are shown in Table 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`

3.4 Interaction of `Bitwalker_Peek` and `Bitwalker_Poke`

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

3.4.1 Formal Specification with ACSL and C

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

Figure 3.5 shows the straightforward implementation of the first helper function along with its ACSL contract. The contract contains a lot of preconditions because we only specify an interaction for the case where both functions are called in their normal cases. The reader can compare the preconditions with the ones from the contracts of `Bitwalker_Peek` and `Bitwalker_Poke` with respect to the `normal` behaviors. As a postcondition we formulate that `Bitwalker_Peek` reads exactly the value written by `Bitwalker_Poke`.

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

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

Listing 3.5. Specification of interaction when first calling `Bitwalker_Poke`.

Figure 3.6 shows the straightforward implementation of the second auxiliary function along with its ACSL contract. In contrast to the first function, we work with two bit streams. With `Bitwalker_Peek` a certain bit sequence is read out from one bit stream and written via `Bitwalker_Poke` into the other one. Therefore, we formulate as a postconditions that the two bit streams are equal in these certain ranges.

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

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

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

Listing 3.6. Specification of interaction when first calling `Bitwalker_Peek`.

3.4.2 Formal Verification with Frama-C/WP

All postconditions (and the assertion in figure 3.6) are verified. Therefore, we succeeded to prove that the functions `Bitwalker_Peek` and `Bitwalker_Poke` interact correctly with respect to their specifications.

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

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

3.5 Open Issues

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.