OETCS/WP4/D4.3.2

openETCS

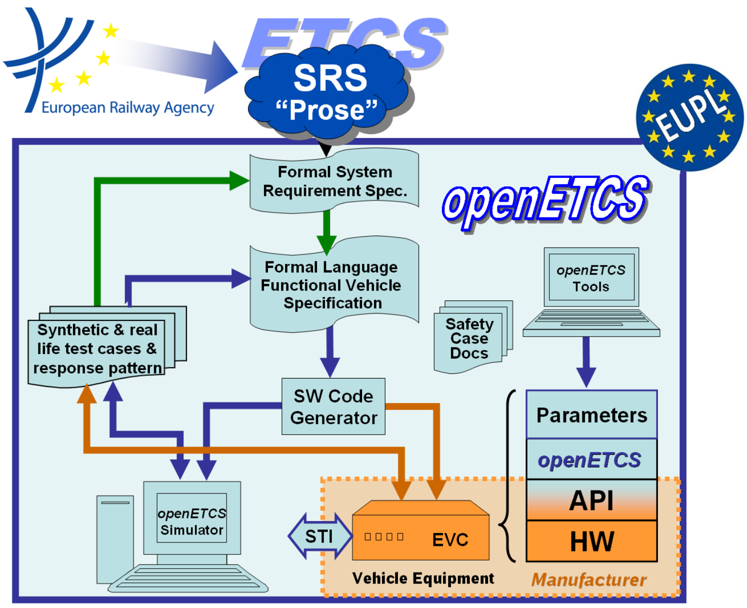
ITEA2 Project
Call 6 11025
2012 – 2015

Work Package 4: “Validation & Verification Strategy”

Final Report on Validation and Verification Report of Implementation/Code

Jens Gerlach and Izaskun de la Torre

March 2015



Funded by:

Federal Ministry of Education and Research
Région de Bruxelles-Capitale
GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, ENERGÍA Y TURISMO

This page is intentionally left blank

Work Package 4: "Validation & Verification Strategy" **OETCS/WP4/D4.3.2**
March 2015

Final Report on Validation and Verification Report of Implementation/Code

Document approbation

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

Jens Gerlach

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

Izaskun de la Torre

Software Quality Systems S.A.

Intermediate report

Prepared for openETCS@ITEA2 Project

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

Table of Contents

Figures and Tables

Figures

Tables

List of code examples

List of Corrections

1 Introduction

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

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

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

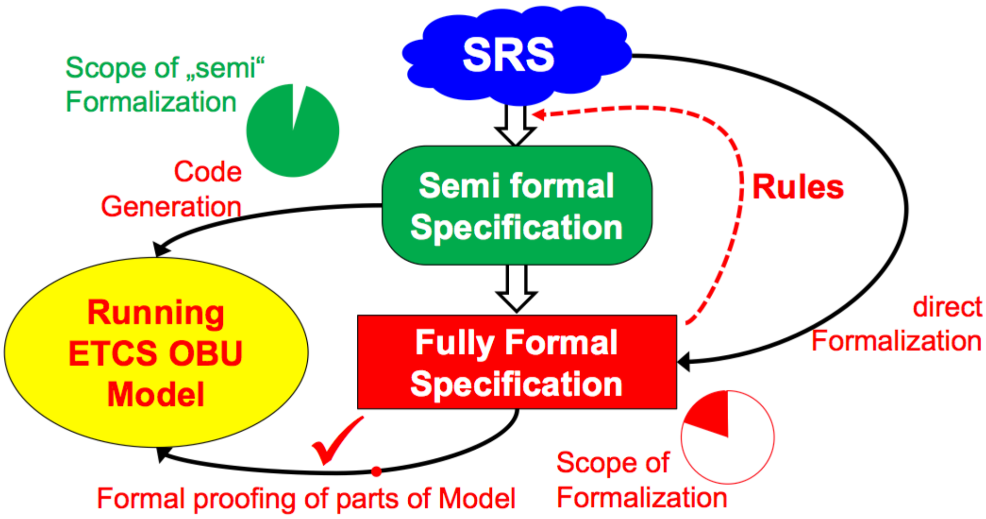


Figure 1.1. Scope of formal methods with in OpenETCS

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

Figure 1.2 shows slightly more detailed the OpenETCS software. The report at hand is limited to the parts encapsulated by C software encapsulated in a dashed box.

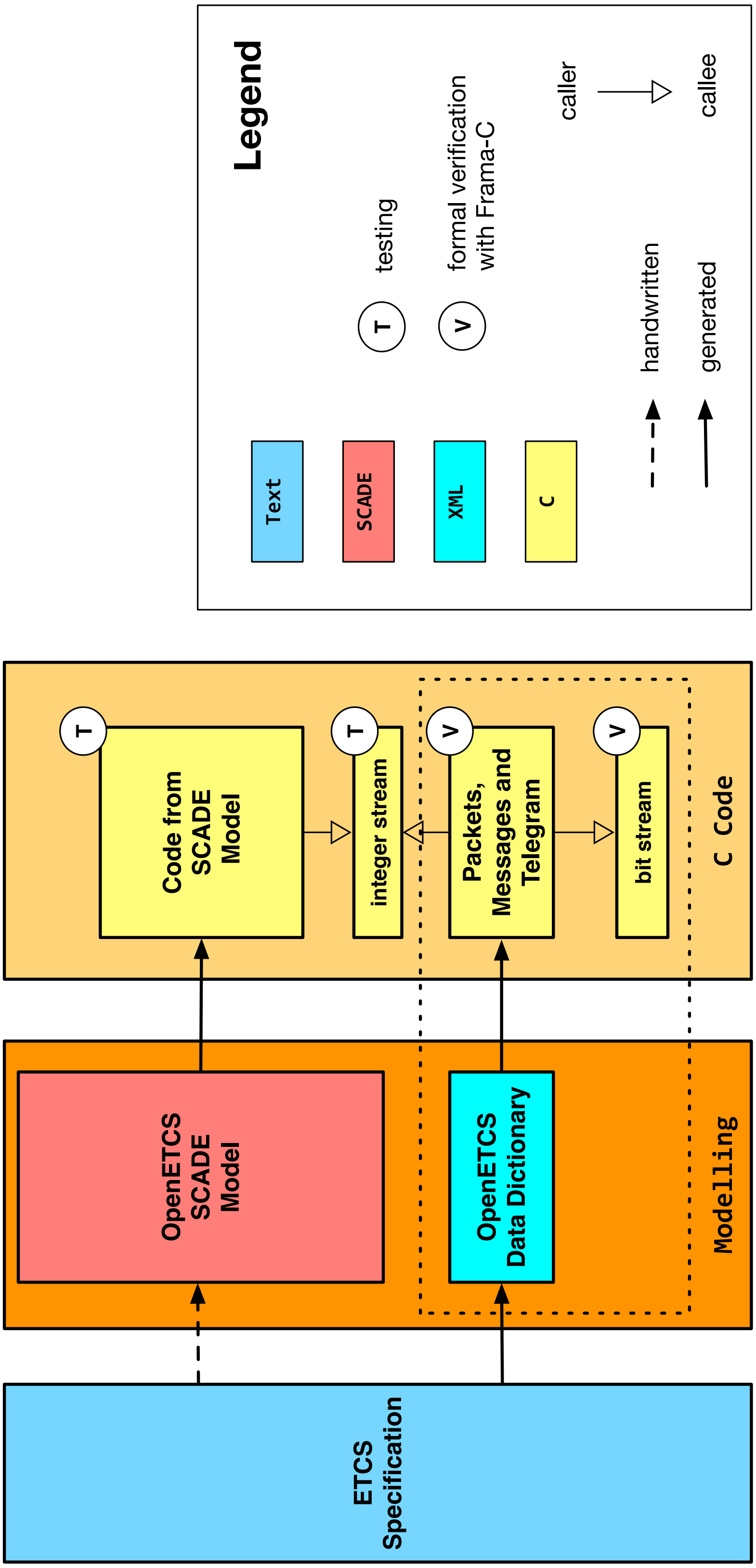


Figure 1.2. Scope of code verification

1.1 Software layers

Figure 1.3 shows the layer structure of the OpenETCS C code. The OpenETCS decoder/enocder is a collection of data structures and associated functions for reading and writing ETCS packets from/to a bit stream.

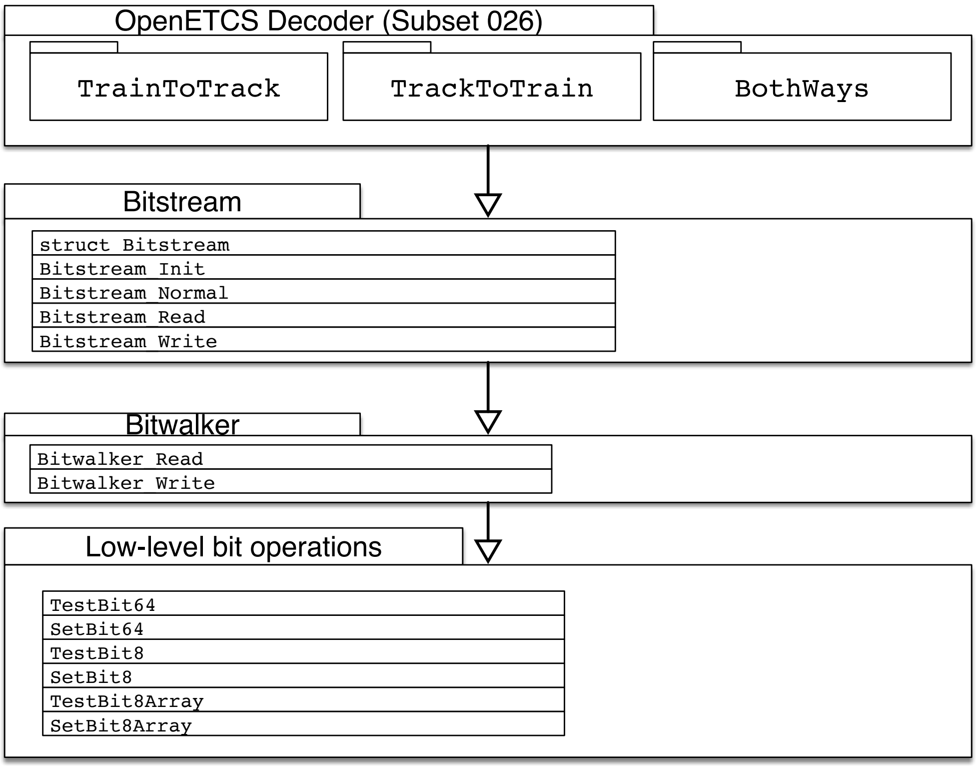


Figure 1.3. Software layers of the OpenETCS C code

In order to fulfill their task the decoder and encoder function rely on an implementation of bit streams in C. The `Bitstream` package in turn is built on top of the so-called *bitwalker* layer. In order to accomplish the task of formal verification of these layers we also provided several functions that read and write individual bits for basic C types.

The main achievement that we present in this report are the results on the formal specification and formal specification of the various software layers in Figure 1.3.

This report is result of the joint work of many OpenETCS partners, notably:

- CEA LIST
- DLR
- Fraunhofer FOKUS
- Siemens
- SQS

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

1. provide evidence that both the generated and a handwritten C code satisfies accepted quality standards
2. develop a formal specification for Subset 026 functionality
3. verify with Frama-C/WP that the software satisfies its formal specification
4. show that the software does not raise runtime errors

1.2 Structure of this document

We represent the C code and related specifications in a top-down approach. Thus, we start on the level of OpenETCS data packets and explain from there the lower software levels.

- Chapter 2 gives a short overview on the Frama-C/WP tool that plays a central role in the verification of OpenETCS C code. Here we also try to rectify some misunderstandings about formal verification that we have encountered in our work.
- Chapter 3 presents the formal specification of OpenETCS packets in ACSL (the specification language of Frama-C). For the sake of an easier understanding we start with the specification of a concrete packet (`AdhesionFactor` in Section 3.1) and explain from there how the other specifications look like.
- The OpenETCS packets are written to and read from bit streams which is represented by the type `Bitstream` and its associated functions. Chapter 4 provides the definition and formal specification of `Bitstream` operations.

  The implementation of `Bitstream` itself relies on lower level bit operations. The formal specification of these operations are presented in Appendix A.
- In Chapter 5, we draw conclusions from this preliminary work and outline the next steps in our verification efforts.

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

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

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

2.1 First steps

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

The mathematical definition of absolute value is very simple

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

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

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

Listing 2.1. An implementation of the absolute value function

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

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

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

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

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

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

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

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

extern int abs_int(int);

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

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

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

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

Listing 2.3. Some simple test cases for `abs_int`

`x`	`abs_int(x)`	Remark
0	0	✓
1	1	✓
10	10	✓
2147483647	2147483647	✓
-1	1	✓
-10	10	✓
-2147483648	-2147483648	$\lightning$

Table 2.1. Test results for `abs_int`

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

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

2.3 Sharpening the contract of `abs_int`

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

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

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

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

```
#include <limits.h>

/*@
    requires x > INT_MIN;

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

Listing 2.4. Taking integer overflows into account

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

There is an important lesson that can be learned here:

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

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

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

```c
#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.

```c
#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.

```c
#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.

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 in C code, *relation chains* with their usual mathematical meaning can be used both in function contracts and assertions.

```c
#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

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.

```c
#include <limits.h>

extern int a;

/*@
    requires x > INT_MIN;

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

Listing 2.11. An implementation with side effects

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

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

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

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:

```
#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`

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

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

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

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

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

```
#include <limits.h>

extern int a;

/*@
    requires x > INT_MIN;

    assigns \nothing; // forbid any side effects

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

Listing 2.13. Specifying the absence of side effects

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

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

#include <limits.h>

extern int a;

/*@
    requires x > INT_MIN;

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

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

Listing 2.14. Finer control of side effects

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

3 ETCS data packets

In the following, we give a top-down presentation of the OpenETCS Decoder software. We discuss the highest, i.e. the data packet level, in this chapter; Chapter 4 elaborates on some intermediate, and Appendix A on the lowest level.

On the data packet level, a total of **FiXme Fatal: ANZAHL** different packets are defined, each by a `struct` declaraion in C. We examplify our discussion on the alphabetically first packet, `AdhesionFactor` (Section 3.1), and give some comments on considerations with respect to other packets (Section 3.2).

In order to cope with the similarity of specification, implemetation, and verification tasks for all packets, we have chosen to automatically generate **FiXme Fatal: Was genau? struct declaraion, contract, ...** from the ETCS Subset 026 requirements description.**FiXme Fatal: Name improvisiert**

3.1 Formal specification of `AdhesionFactor`

3.1.1 `AdhesionFactor` in ETCS

Subset 026 defines package *adhesion factor* (packet 71) as shown in Table 3.1.

variable name	number of bits
NID_PACKET	8
Q_DIR	2
L_PACKET	13
Q_SCALE	2
D_ADHESION	15
L_ADHESION	15
M_ADHESION	1

Table 3.1. Packet `AdhesionFactor` as defined by ETCS

3.1.2 The type `AdhesionFactor`

Listing 3.1 shows the definition of type `AdhesionFactor` as it is generated from the ETCS specification shown in Section 3.1.1.

```
struct AdhesionFactor
{
    PacketHeader header;

    // TransmissionMedia=Any
    // This packet is used when the trackside requests a change of
    // the adhesion factor to be used in the brake model.
    // Packet Number = 71

    uint64_t  Q_DIR;            // # 2
    uint64_t  L_PACKET;         // # 13
    uint64_t  Q_SCALE;          // # 2
    uint64_t  D_ADHESION;       // # 15
    uint64_t  L_ADHESION;       // # 15
    uint64_t  M_ADHESION;       // # 1
};

typedef struct AdhesionFactor AdhesionFactor;
```

Listing 3.1. Defintion of the type `AdhesionFactor`

3.1.3 ACSL predicates `AdhesionFactor`

Listing 3.2 shows the definition of the logic functions `BitSize` and `MaxBitSize` for `AdhesionFactor`. The former function uses a macro that contains the size of `AdhesionFactor` in bits. The functions are used in Listing 3.8 and Listing 3.9 where the overloading of the logic predicates allows for a more generic ACSL contract for the `EncodeBit` and `DecodeBit` functions.

```
/*@
    logic integer BitSize{L}(AdhesionFactor* p) = ADHESIONFACTOR_BITSIZE;

    logic integer MaxBitSize{L}(AdhesionFactor* p) = BitSize(p);
*/
```

Listing 3.2. Definition of the `BitSize` predicates for `AdhesionFactor`

Listing 3.3 shows the definition of the `Invariant` predicate for `AdhesionFactor`. The predicate is the conjunction of the (trivial) `Invariant(uint64_t)` predicates of all members of on object of type `AdhesionFactor`.

```
/*@
    predicate Invariant(AdhesionFactor* p) =
      Invariant(p->Q_DIR)             &&
      Invariant(p->L_PACKET)          &&
      Invariant(p->Q_SCALE)           &&
      Invariant(p->D_ADHESION)        &&
      Invariant(p->L_ADHESION)        &&
      Invariant(p->M_ADHESION);
*/
```

Listing 3.3. Definition of the `Invariant` predicate for `AdhesionFactor`

Listing 3.4 shows the definition of the `UpperBitsNotSet` predicate for `AdhesionFactor`. The predicate `UpperBitsNotSet(AdhesionFactor*)` evaluates to true if and only if the values of all members of `AdhesionFactor` fit into their assigned numbers of bits. This functionality is ensured by the conjunction of the `UpperBitsNotSet(uint64_t, uint32_t)` predicate, which is explained in Appendix A.2, for all members of `AdhesionFactor`.

```
/*@
    predicate UpperBitsNotSet(AdhesionFactor* p) =
      UpperBitsNotSet(p->Q_DIR,            2)   &&
      UpperBitsNotSet(p->L_PACKET,         13)  &&
      UpperBitsNotSet(p->Q_SCALE,          2)   &&
      UpperBitsNotSet(p->D_ADHESION,       15)  &&
      UpperBitsNotSet(p->L_ADHESION,       15)  &&
      UpperBitsNotSet(p->M_ADHESION,       1);
*/
```

Listing 3.4. Definition of the `UpperBitsNotSet` predicate for `AdhesionFactor`

Listing 3.5 shows the definition of predicate `Separated` for `AdhesionFactor`. The predicate `Separated(stream, p)` is true if and only if the two objects `*stream` and `*p` do not overlap in memory. Thus, writing into the stream will not change `*p` and vice versa.

```
/*@
    predicate Separated(Bitstream* stream, AdhesionFactor* p) =
      \separated(stream, p) &&
      \separated(stream->addr + (0..stream->size-1), p);
*/
```

Listing 3.5. Definition of the `Separated` predicate for `AdhesionFactor`

Listing 3.6 shows the definition of the `EqualBits` predicate for `AdhesionFactor`. Based on the ETCS specification, this predicate describes a relationship between the bits of the individual members of an object of type `AdhesionFactor` and those of a bit stream. This predicate will be used to formally describe the transfer of bits from a bit stream to an object of type `AdhesionFactor` and vice versa. The definition of the predicate `EqualBits(AdhesionFactor*)` uses the predicate `EqualBits(uint64_t)`, which is explained in Section 4.1.

```
/*@
    predicate EqualBits(Bitstream* stream, integer pos, AdhesionFactor* p)
        =
      EqualBits(stream, pos,        pos + 2,   p->Q_DIR)            &&
      EqualBits(stream, pos + 2,    pos + 15,  p->L_PACKET)         &&
      EqualBits(stream, pos + 15,   pos + 17,  p->Q_SCALE)          &&
      EqualBits(stream, pos + 17,   pos + 32,  p->D_ADHESION)       &&
      EqualBits(stream, pos + 32,   pos + 47,  p->L_ADHESION)       &&
      EqualBits(stream, pos + 47,   pos + 48,  p->M_ADHESION);
*/
```

Listing 3.6. Definition of the `EqualBits` predicate for `AdhesionFactor`

3.1.4 Formal specification of `AdhesionFactor_UpperBitsNotSet`

Listing 3.7 shows the contract of the `UpperBitsNotSet` function for `AdhesionFactor`. The function contract includes the **requires** clauses, labeled `valid` and **invariant**. These limit the significance of the **ensures** and **assigns** clauses to the `AdhesionFactor` objects that also satisfy the **requires** clauses. The `valid` clause only evaluates to true if the `*p` is a valid pointer. The **invariant** clause requires `Invariant(p)` to evaluate to true. The `Invariant(AdhesionFactor*)` predicate is explained in Section 3.1.3. The contract also inlcudes a statement on the return value of the function, labeled `result`. This clause ensures that the function's return value for `AdhesionFactor* p` matches the evaluation of the predicate `UpperBitsNotSet(p)` from Section 3.1.3. With the **assigns** `\nothing` clause the contract furthermore specifies that this function has no side effects.

```
/*@
    requires valid:       \valid_read(p);
    requires invariant:   Invariant(p);

    assigns \nothing;

    ensures result:  \result <==> UpperBitsNotSet(p);
*/
int AdhesionFactor_UpperBitsNotSet(const AdhesionFactor* p);
```

Listing 3.7. Contract for `UpperBitsNotSet` function of `AdhesionFactor`

3.1.5 Formal specification of `AdhesionFactor_DecodeBit`

Listing 3.8 shows the contract for the `DecodeBit` function for `AdhesionFactor`. The behavior of the function is specified using the two disjoint behaviors `normal_case` and `error_case`. The requirements `valid_stream`, `stream_invariant`, `valid_package` and `separation` apply to both behaviors and limit the set of combinations of input arguments for which the **`ensures`** and **`assigns`** clauses describe the behavior of the function.

The **`assigns`** clauses in the contract's body describe the side effects of the function. If the function contract is split into multiple behaviors, like here, common **`assigns`** clauses, which contain the union of the behavior's **`assigns`** clauses, are needed outside of the behaviors. Their meaning will become clear when discussing the individual behaviors.

For both behaviors the `unchanged` clause states that none of the bits in the bit stream are written by the function.

- `valid_stream` is only met if `Readable(stream)` applies. The `Readable(Bitstream*)` predicate is described in Chapter 4.1.
- `stream_invariant` is only met if `*stream` is long enough to read an object of the length of `AdhesionFactor` from. The `Invariant(Bitstream*, integer)` predicate is introduced in Chapter 4.1.
- `valid_package` is only met if `p` is a valid pointer for read and write operations.
- `separation` requires that `*stream` and `*p` do not overlap in the memory. The `Separated` predicate is introduced in Section 3.1.3.

`normal_case` describes the function's behavior if `*stream` contains enough unread bits to fill all members of `*p`. In this case an object of type `AdhesionFactor` is decoded from the stream and thus `*p` is written. The latter is stated by the first **`assigns`** clause. In this context the **`ensures`** clauses `equal` and `upper` describe the relationship of the bits in the bit stream and the bits of the members of `*p`. Furthermore `stream->bitpos` will be updated. The effects of this operation are described by the second **`assigns`** and the `increment` clauses.

`error_case` describes the function's behavior in the opposite case i.e. if `*stream` is exhausted before all members of `*p` are read. In this case the function has no side effects and especially does not write `*p` or `stream->bitpos`. The **`ensures`** clause `result` states, that the return value of the function equals `0`. In `normal_case` this value is specified to equal `1`.

The distinguishing predicate for the two behaviors is `Normal(Bitstream*, integer)`, which appears in the `assumes` clauses within both behaviors and is explained in Chapter 4.1.

```
/*@
    requires valid_stream:      Readable(stream);
    requires stream_invariant:  Invariant(stream, MaxBitSize(p));
    requires valid_package:     \valid(p);
    requires separation:        Separated(stream, p);

    assigns stream->bitpos;
    assigns *p;

    ensures unchanged:          Unchanged{Here,Old}(stream, 0, 8*stream->
        size);

    behavior normal_case:
      assumes Normal{Pre}(stream, MaxBitSize(p));

      assigns stream->bitpos;
      assigns *p;

      ensures invariant:  Invariant(p);
      ensures result:     \result == 1;
      ensures increment:  stream->bitpos == \old(stream->bitpos) + BitSize(
          p);
      ensures equal:      EqualBits(stream, \old(stream->bitpos), p);
      ensures upper:      UpperBitsNotSet(p);

    behavior error_case:
      assumes !Normal{Pre}(stream, MaxBitSize(p));

      assigns \nothing;

      ensures result: \result == 0;

    complete behaviors;
    disjoint behaviors;
*/
int AdhesionFactor_EncodeBit(const AdhesionFactor* p, Bitstream* stream);
```

Listing 3.8. Contract for `DecodeBit` function of `AdhesionFactor`

3.1.6 Formal specification of `AdhesionFactor_EncodeBit`

```
/*@
    requires valid_stream:      Writeable(stream);
    requires stream_invariant:  Invariant(stream, MaxBitSize(p));
    requires valid_package:     \valid_read(p);
    requires invariant:         Invariant(p);
    requires separation:        Separated(stream, p);

    assigns stream->bitpos;
    assigns stream->addr[0..(stream->size-1)];

    behavior normal_case:
      assumes Normal{Pre}(stream, MaxBitSize(p)) && UpperBitsNotSet{Pre}(p)
          ;

      assigns stream->bitpos;
      assigns stream->addr[0..(stream->size-1)];

      ensures result:     \result == 1;
      ensures increment:  stream->bitpos == \old(stream->bitpos) + BitSize(
          p);
      ensures left:       Unchanged{Here,Old}(stream, 0, \old(stream->
          bitpos));
      ensures middle:     EqualBits(stream, \old(stream->bitpos), p);
      ensures right:      Unchanged{Here,Old}(stream, stream->bitpos, 8 *
          stream->size);

    behavior values_too_big:
      assumes Normal{Pre}(stream, MaxBitSize(p)) && !UpperBitsNotSet{Pre}(p
          );

      assigns \nothing;

      ensures result:        \result == -2;

    behavior invalid_bit_sequence:
      assumes !Normal{Pre}(stream, MaxBitSize(p));

      assigns \nothing;

      ensures result:       \result == -1;

    complete behaviors;
    disjoint behaviors;
*/
int AdhesionFactor_DecodeBit(AdhesionFactor* p, Bitstream* stream);
```

Listing 3.9. Contract for `EncodeBit` function of `AdhesionFactor`

3.1.7 Formal verification of `AdhesionFactor`

3.2 Formal specification of other packets

4 The Bit Stream Layer

In this section, we describe the intermediate abstractions levels the packet level (section 3) relies on. First, we discuss in Section 4.1 a level where operation arguments typically are include a `C` structure `struct bitstream*`, which encapsulates bitstream data and all related administration information. In Section 4.2, we present a level still working on bit sequences, but with an operation typically having one argument for every bitstream data or administration input. Appendix A finally presents even lower levels.

4.1 The `Bitstream` abstraction

The operations on packet data structures were implemented by operations that on a `struct bitstream*` argument. **FiXme Fatal: Hier heisst es "struct bitstream*", in ch.3 heisst es "Bitstream*". Stimmt die Gross-/Kleinschreibung?** The latter are described in this section.

The operation `Bitstream_Read(stream,length)` reads the next `length` bits from the bitstream `stream`, and returns them as a `uint64_t` value. Its formal ACSL specification is shown in Listing 4.1. It requires `stream`

- to point to a valid memory area (requirement "`valid`"),
- to adhere to its data type invariant ("`invariant`"), and
- not to be exhausted ("`normal`").

It is allowed to — and usually in fact will — modify the current bit position within `stream`, but it has to leave all other memory unchanged (expressed by the "`assigns`" clause). After completion of the operation,

- the current bit position has been increased accordingly ("`ensures pos`"),
- the return value equals, bit by bit, the stream between the current bit position on entry and that on exit ("`changed`"),
- in particular, all but the `length` least significant bits[2] of the return value are zero ("`upper`"),
- `stream`'s total size remains unaffected ("`size`"), and
- so do all of its content bits ("`unchanged`").

The formal definitions of the ACSL predicates used in `Bitstream_Read`'s contract are given in Listing 4.3; they build upon the internal details of the `Bitstream` data structure shown in Listing 4.2.

[2] Bit positions are counted differently in Frama-C and in the openETCS project, cf. Figure A1 in Appendix A.2. In this report, we preferably used the terms "least" and "most significant bit(s)" to designate a (range of) bit position(s) indenpendent of the coordinate system.

```
/*@
  requires  valid:      Readable(stream);
  requires  invariant:  Invariant(stream, length);
  requires  normal:     Normal(stream, length);

  assigns   stream->bitpos;

  ensures   pos:        stream->bitpos == \old(stream->bitpos) + length;
  ensures   changed:    EqualBits(stream, \old(stream->bitpos), stream->
      bitpos, \result);
  ensures   upper:      UpperBitsNotSet(\result, length);
  ensures   size:       stream->size == \old(stream->size);
  ensures   unchanged:  Unchanged{Here,Old}(stream, 0, 8 * stream->size);
*/
uint64_t Bitstream_Read(Bitstream* stream, uint32_t length);
```

Listing 4.1. Reading from a bitstream

— Insert drawing —

Figure 4.1. Bit coincidences required by `EqualBits`

- Predicate `Readable` requires that a stream's data area is complete accessible for read.
- Similarly, `Writeable` requires that it is accessible for update.
- Predicate `Invariant` requires that a **`struct`** `Bitstream`'s data area doesn't overlap with the **`struct`** itself, and that some further, lower-level invariant holds (see Section 4.2 below).

  In a similar way, predicates `Normal` and `EqualBits` is reduced to a lower-level predicate of the same name, respectively.[3]
- A clause `Normal(size,bitpos,length)` requires `bitpos` to be such that at least `length` more bits are available beyond it in a stream of byte-size `size`.[4]
- A predicate call `Unchanged{A,B}(stream,first,last)` succeeds if, and only if, all data bits [`first...last`) of `stream` agree in memory state `A` and `B`. For example, it is used with `A` and `B` instantiated to Frama-C's reserved keyword "`Here`" and "`Old`", denoting the memory state after and before operation completion and entry, respectively; cf. Listing 4.6.
- A clause `EqualBits(addr,first,last,value)` requires bits [`first...last`) in the byte array at `addr` to coincide with the corresponding least significant bits of `value`, cf. Figure 4.1.

As a kind of constructor for `Bitstream`, we provide the operation `Bitstream_Init`, shown with its contract in Listing 4.4. Moreover, we provide a test for exhaustion of a `Bitstream`, shown in Listing 4.5.

[3] Frama-C allows for predicate overloading.
[4] We tacitly assume that each stream has a multiple of 8 bits available. **FiXme Fatal: unrealistic!?**

```
struct Bitstream
{
    uint8_t*  addr;     // start address of stream data
    uint32_t  size;     // length of stream data in bytes
    uint32_t  bitpos;   // current bit position within stream data
};
typedef struct Bitstream Bitstream;
```

Listing 4.2. Details for the bitstream data structure

```
/*@
  predicate
    Readable{L}(Bitstream* stream) = \valid(stream) &&
      \valid_read(stream->addr + (0..stream->size-1));

  predicate
    Writeable{L}(Bitstream* stream) = \valid(stream) &&
      \valid(stream->addr + (0..stream->size-1));

  predicate
    Invariant{L}(Bitstream* stream, integer length) =
      \separated(stream, stream->addr + (0..stream->size-1)) &&
      Invariant(stream->size, stream->bitpos, length);

  predicate
    Normal{L}(Bitstream* stream, integer length) =
      Normal(stream->size, stream->bitpos, length);

  predicate
    Unchanged{A,B}(Bitstream* stream, integer first, integer last) =
      \forall integer i;  first <= i < last ==>
        (\at(Bit8Array(stream->addr, i),A) <==>
         \at(Bit8Array(stream->addr, i),B));

  predicate
    EqualBits{A}(Bitstream* stream, integer first, integer last, uint64_t
        value) =
      EqualBits{A}(stream->addr, first, last, value);

*/
```

Listing 4.3. ACSL predicates used in bitstream layer contracts

Listing 4.6 shows contract of `Bitstream_Write`, and moreover exemplfies its implementation.

Most parts of the contract are quite similar to that of `Bitstream_Read` in Listing 4.1. Differences are the following:

- We require that the `value` to be written fits into the specified `length`, i.e. its unused most significant bits are zero (requirement "`upper`").
- The operation is allowed to change the contents of the bitstream (first `assigns` clause) in addition to the streams current bit position (second `assigns` clause), but no other memory locations.
- Since we couldn't specify in the `assigns` clauses which bits exactly are allowed to be modified, we give the details in two `ensures` clauses named "`unchanged`": All bits before the stream's `bitpos` on operation entry, and after its `bitpos` on exit, must remain unchanged.

```
/*@
  requires valid:     Writeable(stream);
  requires bit_size:  8 * size <= UINT32_MAX;
  requires valid_pos: bitpos <= 8 * size;
  requires separated: \separated(addr + (0..size-1), stream);

  assigns  stream->addr, stream->size, stream->bitpos;

  ensures  addr:      stream->addr == addr;
  ensures  size:      stream->size == size;
  ensures  bitpos:    stream->bitpos == bitpos;
  ensures  invariant: Invariant(stream, 0);
*/
void Bitstream_Init(Bitstream* stream, uint8_t* addr, uint32_t size,
    uint32_t bitpos);
```

Listing 4.4. Setting-up a bitstream

```
/*@
  requires valid:     Readable(stream);
  requires invariant: Invariant(stream, length);

  assigns \nothing;

  ensures  result:    \result <==> Normal(stream, length);
*/
int Bitstream_Normal(const Bitstream* stream, uint32_t length);
```

Listing 4.5. Testing a bitstream for exhaustion

The implementation just employs the lower-level operation `Bitwalker_Write` to write the bits, and appropriately updates the `stream`'s `bitpos`. Two assertions were needed to help the provers establishing that `value`'s bits are actually written to `stream`'s data array by `Bitwalker_Write`, and that they aren't destoyed during `bitpos` update.

```
/*@
  requires valid:      Writeable(stream);
  requires invariant:  Invariant(stream, length);
  requires normal:     Normal(stream, length);
  requires upper:      UpperBitsNotSet(value, length);

  assigns stream->addr[0..stream->size - 1];
  assigns stream->bitpos;

  ensures  pos:        stream->bitpos == \old(stream->bitpos) + length;
  ensures  changed:    EqualBits(stream, \old(stream->bitpos), stream->
     bitpos, value);
  ensures  unchanged:  Unchanged{Here,Old}(stream, 0, \old(stream->bitpos))
     ;
  ensures  unchanged:  Unchanged{Here,Old}(stream, stream->bitpos, 8 *
     stream->size);
  ensures  size:       stream->size == \old(stream->size);
*/
void Bitstream_Write(Bitstream* stream, uint32_t length, uint64_t value)
{
    Bitwalker_Write(stream->addr, stream->size, stream->bitpos, length,
       value);
    //@ assert EqualBits(stream, stream->bitpos, stream->bitpos + length,
       value);

    stream->bitpos += length;
    //@ assert EqualBits(stream, \at(stream->bitpos,Pre), stream->bitpos,
       value);
}
```

Listing 4.6. Writing to a bitstream

4.2 Reading and writing bit sequences

In this section, we describe the operations that handle plain bit sequences. They are used to implement the `struct bitstream*` operations for Section 4.1.

The operation `Bitwalker_Read(addr,size,bitpos,length)` reads `length` bits starting at `bitpos` from the array `addr` of byte-size `size`, and returns them as a `uint64_t` value. Its ACSL contract is shown in Listing 4.7. It requires

- all bytes of the `addr` array to be accessible for read (requirement “`valid`”),
- some data type invariants to hold (“`invariant`”), viz.
  - the total number of array bits to fit into a `uint32_t`,
  - the result value to fit into a `uint64_t`,
  - the end bit position — and hence also the start bit position `bitpos` and `length` — to fit into a `uint32_t`,

  and
- the bit range [`bitpos...bitpos+length`) to fit into the array at `addr` (“`normal`”).

The operation is not allowed to modify any (non-local) memory (expressed by the “`assigns`” clause). After completion of the operation,

- the return value shall coincide bit by bit with the specified range of the `addr` array (“`ensures equal`”), and
- in particular, all but the least significant `length` bits of the return value shall be zero (“`upper`”).

```
/*@
  requires  valid:      Readable(addr, size);
  requires  invariant:  Invariant(size, bitpos, length);
  requires  normal:     Normal(size, bitpos, length);

  assigns   \nothing;

  ensures   equal:      EqualBits(addr, bitpos, bitpos + length, \result);
  ensures   upper:      UpperBitsNotSet(\result, length);
*/
uint64_t Bitwalker_Read(uint8_t* addr, uint32_t size, uint32_t bitpos,
    uint32_t length);
```

Listing 4.7. Reading a bit sequence

The formal definitions of the used ACSL predicates are given in Listing 4.8. Again, the tacit assumption that the array contains sensible data upto its very last bit is used in predicate `Normal`.

```
/*@
   predicate Readable{L}(uint8_t* addr, integer size) = \valid_read(addr +
       (0..size-1));

   predicate Writeable{L}(uint8_t* addr, integer size) = \valid(addr + (0..
       size-1));

   predicate Invariant{L}(integer size, integer bitpos, integer length) =
       8 * size <= UINT32_MAX          &&
       length <= 64                    &&
       bitpos + length <= UINT32_MAX;

    predicate Normal{L}(integer size, integer bitpos, integer length) =
       bitpos + length <= 8 * size;
*/
```

Listing 4.8. ACSL predicates used in bitsequence layer contracts

Listing 4.9 shows the contract, and the implementation, of `Bitwalker_Write`. The following peculiarities are observed when the former is compared to `Bitwalker_Read`’s contract:

- We require that the `value` to be written fits into the specifiec `length`, i.e. all but its `length` least significant bits are zero (requirement “`upper`”).
- The operation may modify the data array at `addr`, but nothing else.
- Again, we give the details of which data bits exactly are allowed to be changed in two `ensures` clauses, named “`left`” and “`right`”, and requiring all bits before `bitpos` and after `bitpos+length` to remain unchanged, respectively.

In the implementation, which is shown here as an example, we used the straight-forward algorithm that takes a bit from `value` and places it into the `addr` array, bit by bit. In order for the provers to establish that algorithm’s correctness, we had to provide a total of six ACSL clauses about the loop:

- The loop variable, `i`, always ranges in the interval [`bitpos...bitpos+length`] — loop invariant "`bound`". Note that the highest value is actually taken, viz. on exit of the loop body in the last iteration, subsequently causing the loop to terminate.
- The bits before `bitpos`, and after `bitpos+length` remain as they were on operation entry — invariant "`left`" and "`right`", respectively.
- In the `i`th iteration, the bits [`bitpos...bitpos+i`) agree with the least significant `i` bits of `value` — invariant "`middle`".
- The loop code is allowed to modify the variable `i`, and the whole array at `addr`, but nothing else — `loop assigns` clause.
- The value of the integer expression `bitpos+length-i` is non-negative thoughout the whole loop execution, but is decreased in every iteration — `loop variant` clause. Therefore, the loop is guaranteed to terminate eventually.

```
/*@
  requires valid:      Writeable(addr, size);
  requires invariant:  Invariant(size, bitpos, length);
  requires normal:     Normal(size, bitpos, length);
  requires upper:      UpperBitsNotSet(value, length);

  assigns addr[0..size-1];

  ensures  left:       Unchanged{Here,Old}(addr, 0, bitpos);
  ensures  middle:     EqualBits(addr, bitpos, bitpos + length, value);
  ensures  right:      Unchanged{Here,Old}(addr, bitpos + length, 8 * size)
      ;
*/
void Bitwalker_Write(uint8_t* addr, uint32_t size, uint32_t bitpos,
    uint32_t length, uint64_t value);
{
    /*@
      loop invariant bound:   bitpos <= i <= bitpos + length;
      loop invariant left:    Unchanged{Here,Pre}(addr, 0, bitpos);
      loop invariant middle:  EqualBits(addr, bitpos, i, value, length);
      loop invariant right:   Unchanged{Here,Pre}(addr, i, 8 * size);

      loop assigns  i, addr[0..size-1];
      loop variant  bitpos + length - i;
    */
    for (uint32_t i = bitpos; i < bitpos + length; ++i)
    {
        int flag = TestBit64(value, (64 - length) + (i - bitpos));
        SetBit8Array(addr, size, i, flag);
    }
}
```

Listing 4.9. Writing a bit sequence

The implementation of the bit sequence operations is based on operations to write and to read a single bit. The details of the latter, as well as of the predicates used in their contracts, are given in Appendix A.

5 Conclusion

Appendix A: Low-level bitstream operations

In this appendix, we describe the implementation of the low-level bitstream operations. They were used to implement the bit sequence abstraction level, cf. Section 4.2. Since a write operation moves bits from a `uint64_t` value into an array of `uint8_t` values, and a read operations moved them the other way round, we need bit operations on both data types. They are given in Subsection A.1.1 (array of `uint8_t`), A.1.2 (single `uint8_t`), and A.1.3 (single `uint64_t`) below.

A.1 Reading and writing individual bits

A.1.1 8 bit arrays

The operation `TestBit8Array(addr,size,pos)` returns the `pos`th bit within the array at `addr` of byte-size `size`.[5] Its contract and its implementation is shown in Listing A.1. See Listing A.8 for the definition of the predicate `Bit8Array`. The array bits are counted starting with the most significant one of the first byte, cf. Figure A1 below. `TestBit8(bytevalue,bitadr)` returns the `bitadr`th bit within `bytevalue`, its is discussed in Appendix A.1.2 below.

```
/*@
    requires valid:  \valid_read(addr + (0..size-1));
    requires size:   8 * size <= UINT32_MAX;
    requires pos:    pos < 8 * size;

    assigns \nothing;

    ensures result:  \result != 0 <==> Bit8Array(addr, pos);
*/
static inline int TestBit8Array(uint8_t*  addr, uint32_t size, uint32_t pos
    )
{
    return TestBit8(addr[pos / 8], pos % 8);
}
```

Listing A.1. Reading a bit of an `uint8_t` array

Similarly, the operation `SetBit8Array(addr,size,pos,flag)` sets the `pos`th bit within the array at `addr` of byte-size `size` to `flag`. Its contract is shown in Listing A.2. It requires

- the whole array to be accessible for update (requirement "`valid`"),
- each possible bit position in the array to fit into a `uint32_t` ("`size`"), and
- the given `pos` to be a valid bit position in the array ("`pos`").

The `assigns` clause allows the operation to change the contents of the array, but no other memory locations. On completion, the operation shall guarantee

[5] This parameter isn't actually used in the code, but merely in the contract.

- that the value of `flag`[6] is actually stored at the designated bit position ("`ensures middle`"), and
- that all other bits remain unchanged ("`left`", "`right`").

Two fairly sophisicated hints had to be provided as assertions in the body in order for the provers to establish the contract's post-conditions.

```
/*@
    requires valid:  \valid(addr + (0..size-1));
    requires size:   8 * size <= UINT32_MAX;
    requires pos:    pos < 8 * size;

    assigns addr[0..size-1];

    ensures left:   Unchanged{Here,Old}(addr, 0, pos);
    ensures middle: Bit8Array(addr, pos) <==> (flag != 0);
    ensures right:  Unchanged{Here,Old}(addr, pos + 1, 8 * size);
*/
static inline void SetBit8Array(uint8_t* addr, uint32_t size, uint32_t pos,
     int flag)
{
    uint32_t i = pos / 8u;
    uint32_t k = pos % 8u;

    addr[i] = SetBit8(addr[i], k, flag);

    // The following assertion claims that in byte with index "pos/8"
    // the bits with indices different from "k" do not change
    /*@
      assert bits_in_byte:
        \forall integer j; (0 <= j < 8  && j != k) ==>
        (Bit8(addr[pos/8], j) <==> \at(Bit8(addr[pos/8], j), Pre));
    */

    // The following assertion claims that in every byte
    // with an index that is different from "pos/8" no bit is changed.

    /*@
        assert other_bytes:
        \forall integer l, j; (0 <= l < size  &&  l != pos/8  &&  0 <= j <
            8) ==>
          (Bit8(addr[l], j) <==> \at(Bit8(addr[l], j), Pre));
    */

}
```

Listing A.2. Writing a bit of an `uint8_t` array

A.1.2 8 bits

The operation `TestBit8(value,pos)` returns the `pos`$^{\text{th}}$ bit of `value`. Its contract is shown in Listing A.3. `pos` must not exceed 7 (requirement "`pre`"), no memory may be modified (`assigns`), and the result is non-zero if, and only if, the specified bit is set. The shown implementation additionally guarantees that the result is zero or one, which is not specified in the contract since this property isn't needed. Returning just `flag` rather than `flag!=0u` would satisfy the contract also, and would be slightly faster.

[6] Any non-zero `flag` value is treated like 1. This is ensured by the contract of the called operation `SetBit8`, cf. Appendixsubsec:low-level 8.

```
/*@
    requires pre:  pos < 8;

    assigns \nothing;

    ensures pos:  \result != 0 <==> Bit8(value, pos);
*/
static inline int TestBit8(uint8_t value, uint32_t pos)
{
    uint8_t mask = ((uint8_t) 1) << (7u - pos);
    uint8_t flag = value & mask;

    return flag != 0;
}
```

Listing A.3. Reading a bit of `uint8_t`

Dual to `TestBit8`, the operation `SetBit8(value,pos,flag)` returns `value`, with the `pos`th bit set to `flag`. Its contract is shown in Listing A.4. Again, `pos` mustn't exceed 7 (requirement "`pre`"), no memory may be modified (`assigns` clause), the return value coincides with `value`, except possibly at `pos` (ensures "`left`" and "`right`"), and `flag` is written to the approriate bit of `value` ("`pos`"). The implementation branches on the value of `flag`, and clears or sets the appropriate bit in the usual way. Note that both our contract and our implementation enable us to set a bit by supplying a `flag` value of e.g. 2, whereas the code `mask=flag<<(7-pos); return (value&~mask)|mask;` does not.

```
/*@
    requires pre: pos < 8;

    assigns \nothing;

    ensures left:   EqualBits8(\result, value, 0,   pos);
    ensures pos:    Bit8(\result, pos) <==> (flag != 0);
    ensures right:  EqualBits8(\result, value, pos + 1,  8);
*/
static inline uint8_t SetBit8(uint8_t value, uint32_t pos, int flag)
{
    uint8_t mask = ((uint8_t) 1) << (7u - pos);

    return (flag == 0) ? (value & ~mask) : (value | mask);
}
```

Listing A.4. Writing a bit of `uint8_t`

A.1.3 64 bits

The operations to read and write a bit of a `uint64_t` are closely similar to those working on a `uint8_t`. They are shown in Listing A.5 and A.6 without repeating the comments given in Appendix A.1.2 for the 8 bit version; see in particular Listing A.8 for the employed ACSL predicates.

As a redundant postcondition of the `SetBit64` operation in Listing A.6, "`upper`" guarantees that the leading zeros in `value` are kept in the result, up to, but excluding position `pos`. It was needed to enable the provers to verify code that uses `SetBit64`.

```
/*@
   requires pre: pos < 64;

   assigns \nothing;

   ensures set_bit: \result != 0 <==> Bit64(value, pos);
*/
int TestBit64(uint64_t value, uint32_t pos)
{
    uint64_t mask = ((uint64_t) 1) << (63u - pos);
    uint64_t flag = value & mask;

    return flag != 0u;
}
```

Listing A.5. Reading a bit of `uint64_t`

```
/*@
    requires pre: pos < 64;

    assigns \nothing;

    ensures left:     EqualBits64(\result, value, 0,  pos);
    ensures set_bit:  flag != 0  <==>  Bit64(\result, pos);
    ensures right:    EqualBits64(\result, value, pos + 1,  64);
    ensures upper:    \forall integer i; i >= 64 - pos ==>
                        (UpperBitsNotSet(value, i) ==> UpperBitsNotSet(\
                           result, i));
*/
uint64_t SetBit64(uint64_t value, uint32_t pos, int flag)
{
    uint64_t mask = ((uint64_t) 1u) << (63 - pos);

    return (flag == 0) ? (value & ~mask) : (value | mask);
}
```

Listing A.6. Writing a bit of `uint64_t`

The operation `UpperBitsNotSet64(value,length)` succeeds, i.e. returns a non-zero value, if, and only if, all bits of `value` except the least significant `length` ones are zero. It is used for **FiXme Fatal: ja, wofuer denn eigentlich?**

```
/*@
    requires pre: length <= 64;

    assigns \nothing;

    ensures  not_set: \result <==> UpperBitsNotSet(value, length);
*/
int UpperBitsNotSet64(uint64_t value, uint32_t length);
```

Listing A.7. Test that upper bits are not set

A.2 Formalization of bit operations in Frama-C

The definition of predicate `Bit8` is shown in Listing A.8. It relies on the Frama-C library predicate `BitTest`, performing a coordinate transformation to fit Frama-C's notion of bit positions with the OpenETCS project's notion, cf. Figure A1.

```
/*@
   predicate Bit8{A}(uint8_t v, integer n)  = BitTest(v, 7 - n);

   predicate Bit64{A}(uint64_t v, integer n) = BitTest(v, 63 - n);

   predicate Bit8Array{A}(uint8_t* a, integer n) = Bit8(a[n / 8],n % 8);
*/
```

Listing A.8. Definition of bit test predicates

— Insert drawing —

Figure A1. Bit coordinates in Frama-C and in the OpenETCS project

The predicate `UpperBitsNotSet(value,length)` succeeds if, and only if, all but possibly the least significant `length` bits of `value` are zero. Its definition is shown in Listing A.9.

```
predicate
  UpperBitsNotSet{A}(integer value, integer length) =
    \forall integer i; length <= i ==> !BitTest(value, i);
```

Listing A.9. Definition of the low-level predicate `UpperBitsNotSet`

The predicate `BitTest(v,n)` succeeds if, and only if, the $\mathtt{n}^{\text{th}}$ bit is set in the binary representation of `v`, i.e. iff the truncating integer division of `v` by $2^{\mathtt{n}}$ yields an odd number. **FiXme Fatal: Is there any documentation available? What about negative numbers?** This predicate comes with the standard library of the Frama-C system. Its declaration is shown in Listing A.10.

```
/*@
   predicate   BitTest(integer v, integer n);
*/
```

Listing A.10. The Frama-C library predicate `BitTest`

Appendix: References

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

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

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

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

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

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

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

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

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

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

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

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

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

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