Work Package 4: “Validation & Verification Strategy”


J. Gerlach, J. Burghardt, T. Lapawczyk and M. Behrens

December 2015
This page is intentionally left blank

Document approbation

<table>
<thead>
<tr>
<th>Lead author:</th>
<th>Technical assessor:</th>
<th>Quality assessor:</th>
<th>Project lead:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jens Gerlach</td>
<td>Virgile Prevosto</td>
<td>Abdelnasir Mohamed</td>
<td>Klaus-Rüdiger Hase</td>
</tr>
<tr>
<td>(Fraunhofer FOKUS)</td>
<td>(CEA LIST)</td>
<td>(AEbt)</td>
<td>(DB Netz)</td>
</tr>
</tbody>
</table>

J. Gerlach, J. Burghardt, T. Lapawczyk
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
jens.gerlach@fokus.fraunhofer.de
www.fokus.fraunhofer.de

M. Behrens
DLR Institute of Transportation Systems
Lilienthalplatz 7
Braunschweig 38108, Germany
www.dlr.de/ts

Intermediate report

Prepared for openETCS@ITEA2 Project

This work is licensed under the "openETCS Open License Terms" (oOLT).
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 | ........................................................................................................ | v |
| List of code examples | ....................................................................................................... | vi |
| 1 Introduction | .......................................................................................................... | 1 |
| 1.1 Software layers | .................................................................................................. | 3 |
| 1.2 Structure of this document | ........................................................................ | 4 |
| 2 An introduction to formal verification with Frama-C/WP | .................................................................................. | 7 |
| 2.1 First steps | ........................................................................................................ | 8 |
| 2.2 Why can Frama-C/WP not verify such a simple function? | ........................................................................ | 8 |
| 2.3 Sharpening the contract of abs_int | ................................................................... | 9 |
| 2.4 Separating specification and implementation | .................................................................. | 11 |
| 2.5 Modular verification | ................................................................................ | 11 |
| 2.6 Dealing with side effects | ........................................................................... | 13 |
| 3 ETCS data packets | .................................................................................................... | 17 |
| 3.1 Overview and classification of data packets | .................................................................... | 17 |
| 3.2 Formal specification of AdhesionFactor | ................................................................... | 20 |
| 3.2.1 AdhesionFactor in ETCS | .......................................................................... | 20 |
| 3.2.2 The type AdhesionFactor | ........................................................................... | 20 |
| 3.2.3 ACSL predicates AdhesionFactor | .................................................................... | 21 |
| 3.2.4 Formal specification of AdhesionFactor_UpperBitsNotSet | ........................................... | 23 |
| 3.2.5 Formal specification of AdhesionFactor_DecodeBit | ............................................... | 24 |
| 3.2.6 Formal specification of AdhesionFactor_EncodeBit | ............................................. | 26 |
| 3.3 Formal specification of other packets | ........................................................................ | 28 |
| 4 The bit stream layer | ............................................................................................... | 29 |
| 4.1 The Bitstream abstraction | ................................................................................ | 29 |
| 4.2 Auxiliary Bitstream functions | .......................................................................... | 32 |
| 4.3 Writing bit sequences | ................................................................................ | 33 |
| 4.4 Reading bit sequences | ................................................................................ | 36 |
| 4.5 Verification of the Bitstream abstraction | .................................................................... | 38 |
| 4.5.1 The verification function Bitstream_WriteThenRead | .................................................. | 38 |
| 4.5.2 The verification function Bitstream_ReadThenWrite | ................................................ | 40 |
| 5 Low-level bitstream operations | .................................................................................. | 43 |
| 5.1 Reading and writing individual bits | ........................................................................ | 43 |
| 5.1.1 Reading and writing 8 bit arrays | ................................................................... | 43 |
| 5.1.2 Reading and writing 8 bit words | ..................................................................... | 45 |
| 5.1.3 Reading and writing 64 bit words | ................................................................... | 46 |
| 5.2 Formalization of bit operations in Frama-C | .................................................................. | 48 |
| 6 Formal verification with Frama-C/WP | .......................................................................... | 51 |
| 6.1 Verification of bit stream and lower-level bit operations | .......................................... | 52 |
| 6.2 Verification of static ETCS packets | ....................................................................... | 52 |
| 7 Verification of integration | ................................................................................................ | 55 |
Figures and Tables

Figures

Figure 1.1. Scope of formal methods with in OpenETCS .................................................................... 1
Figure 1.2. Scope of code verification ............................................................................................ 2
Figure 1.3. Software layers of the OpenETCS C code ....................................................................... 3
Figure 3.1. Relationship between bit stream and packet elements ....................................................... 21
Figure 4.1. Bit coincidences required by EqualBits ........................................................................... 30
Figure 4.2. Sketch of Bitstream_Write ............................................................................................. 33
Figure 4.3. Sketch of Bitstream_Read .............................................................................................. 36
Figure 4.4. Outline of Bitstream_WriteThenRead ........................................................................... 38
Figure 4.5. Outline of Bitstream_ReadThenWrite ............................................................................ 40
Figure 7.1. Overview on integration interfaces ................................................................................. 56

Tables

Table 1.1. Properties to be addressed by interface specification according to EN 50128 ........................................ 4
Table 2.1. Test results for abs_int .................................................................................................. 9
Table 3.1. TrackToTrain packets part 1 .......................................................................................... 18
Table 3.2. TrackToTrain packets part 2 .......................................................................................... 19
Table 3.3. TrainToTrack packets .................................................................................................. 19
Table 3.4. Both-Ways packets ...................................................................................................... 19
Table 3.5. Packet AdhesionFactor as defined by ETCS .................................................................. 20
Table 6.1. Verification results for bit stream and lower-level bit operations ........................................ 52
Table 6.2. Verification results for static track to train packets part 1 ................................................ 52
Table 6.3. Verification results for static track to train packets part 2 ................................................ 53
Table 6.4. Verification results for static train to track packets ........................................................ 53
Table 6.5. Verification results for both-ways packets ...................................................................... 53
Table 8.1. Selection of Frama-C issues identified within the OpenETCS project ................................ 58
# List of code examples

2.1 An implementation of the absolute value function ............................................ 8  
2.2 A first attempt to formally specify abs_int .................................................. 8  
2.3 Some simple test cases for abs_int ............................................................ 9  
2.4 Taking integer overflows into account .......................................................... 10  
2.5 Minimal contract to ensure the absence of runtime errors in abs_int .......... 11  
2.6 Specifying a function prototype in a header file .............................................. 11  
2.7 Implementation at a different location than the specification ...................... 11  
2.8 A simple example of modular verification .................................................... 12  
2.9 Another example of modular verification ...................................................... 12  
2.10 A more complex example of modular verification ....................................... 13  
2.11 An implementation with side effects .......................................................... 13  
2.12 Calling a logging function from abs_int ................................................... 14  
2.13 Specifying the absence of side effects ........................................................ 15  
2.14 Finer control of side effects ....................................................................... 15  
3.1 Definition of the type AdhesionFactor ...................................................... 20  
3.2 Definition of the BitSize predicates for AdhesionFactor ......................... 21  
3.3 Definition of the Invariant predicate for AdhesionFactor ....................... 22  
3.4 Definition of the UpperBitsNotSet predicate for AdhesionFactor .......... 22  
3.5 Definition of the Separated predicate for AdhesionFactor ..................... 22  
3.6 Definition of the EqualBits predicate for AdhesionFactor ..................... 23  
3.7 Contract for UpperBitsNotSet function of AdhesionFactor .................. 23  
3.8 Contract for DecodeBit function of AdhesionFactor ............................... 24  
3.9 Contract for EncodeBit function of AdhesionFactor ............................... 26  
4.1 Details for the Bitstream data structure ................................................... 29  
4.2 Reading from a bitstream .......................................................................... 30  
4.3 ACSL predicates used in bitstream layer contracts ....................................... 31  
4.4 Setting-up a bitstream .............................................................................. 32  
4.5 Testing a bitstream for exhaustion .............................................................. 32  
4.6 Writing to a bitstream ............................................................................... 33  
4.7 ACSL predicates used in bit sequence layer contracts ............................... 34  
4.8 Writing a bit sequence ............................................................................... 35  
4.9 Reading a bit sequence ............................................................................... 36  
4.10 Verifying the scenario “write, then read” .................................................... 39  
4.11 Verifying the scenario “read, then write” ................................................... 40  
5.1 Reading a bit of an uint8_t array ............................................................. 43  
5.2 Writing a bit of an uint8_t array ............................................................... 44  
5.3 Reading a bit of uint8_t ............................................................................ 45  
5.4 Writing a bit of uint8_t ............................................................................ 46  
5.5 Reading a bit of uint64_t .......................................................................... 46  
5.6 Writing a bit of uint64_t .......................................................................... 47  
5.7 Test that upper bits are not set .................................................................. 47  
5.8 Definition of bit test predicates .................................................................. 48  
5.9 Definition of the low-level predicate UpperBitsNotSet ......................... 48  
5.10 Definition of the low-level predicate EqualBits64 ..................................... 48  
5.11 ACSL axioms used in 64-bit contracts ....................................................... 49  
5.12 The Frama-C library predicate BitTest .................................................... 49

This work is licensed under the "openETCS Open License Terms" (oOLT).
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 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’.

---

This work is licensed under the "openETCS Open License Terms" (oOLT).
Figure 1.2. Scope of code verification

This work is licensed under the "openETCS Open License Terms" (oOLT).
1.1 Software layers

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

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

The European standard for railway software [1] § 7.3.4.19 mandates that the specification of software interfaces shall address various properties. Table 1.1 list these properties and also indicates to what extend Frama-C can be used to formally express them.

<table>
<thead>
<tr>
<th>Property</th>
<th>Specification through Frama-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre and post conditions</td>
<td>yes</td>
</tr>
<tr>
<td>definition and description of all boundary values for all specified data</td>
<td>yes</td>
</tr>
<tr>
<td>behaviour when the boundary value is exceeded</td>
<td>yes</td>
</tr>
<tr>
<td>behaviour when the value is at the boundary</td>
<td>yes</td>
</tr>
<tr>
<td>time-critical input and output data</td>
<td>no</td>
</tr>
<tr>
<td>allocated memory for the interface buffers and the mechanisms to detect that the memory cannot be allocated or all buffers are full, where applicable</td>
<td>yes</td>
</tr>
<tr>
<td>existence of synchronization mechanisms between functions</td>
<td>no</td>
</tr>
<tr>
<td>definition and description of all equivalence classes for all specified data and each software function using them</td>
<td>yes</td>
</tr>
<tr>
<td>definition of unused or forbidden equivalence classes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 1.1. Properties to be addressed by interface specification according to EN 50128

We see from this table that Frama-C is a well-justified choice for the specification of railway software.

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

Or to be more precise “ACSL” (ANSI/ISO-C Specification Language) which is the specification language of Frama-C.
of a concrete packet (AdhesionFactor in Section 3.2) 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 Chapter 5.

- Chapter 6 lists results of the formal verification with Frama-C/WP.

- Chapter 7 describes the integration activities with respect to the underlying communication system and the SCADE model.

- In Chapter 8, we draw conclusions from this 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 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 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 $\text{abs}$ in C library we use the name $\text{abs\_int}$.

The mathematical definition of absolute value is very simple

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

A straightforward implementation of $\text{abs\_int}$ is shown in Listing 2.1.

```c
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 $\text{abs\_int}$ that is based on the mathematical definition of the function $x \mapsto |x|$ in Equation 1.

```c
/*@    
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 $\text{abs\_int}$

The first thing to note is that ACSL specifications—or contracts—are placed in special C (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 $\text{abs\_int}$. The ACSL reserved word $\\\result$ is used to refer to the return value of a C. Note that we use the usual C == 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 $\text{abs\_int}$. Listing 2.3 shows our test code whose output is listed in Table 2.1.
```c
#include <stdio.h>
#include <limits.h>

extern int abs_int(int);

void print_abs(int x)
{
    printf("%12d	%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`

<table>
<thead>
<tr>
<th>x</th>
<th>abs_int(x)</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>✓</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483647</td>
<td>✓</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>-10</td>
<td>10</td>
<td>✓</td>
</tr>
<tr>
<td>-2147483648</td>
<td>-2147483648</td>
<td>✗</td>
</tr>
</tbody>
</table>

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 clearly states that the absolute value of INT_MIN is undefined\footnote{See \url{http://www.gnu.org/software/libc/manual/html_node/Absolute-Value.html}}. 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\ref{lst:2.4} shows an extended contract of abs_int that takes the limitations of the type int into account.

```c
#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\ref{lst: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\ref{lst:2.5}, Frama-C/WP can verify that no runtime error will occur.
```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 \texttt{abs\_int} is used. More precisely, we include in Listing 2.8 the header file from Listing 2.6 which contains an AC SL specification of \texttt{abs\_int}.

```c
#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 \texttt{abs\_int} are satisfied. Based on the specification of \texttt{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 \texttt{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 \texttt{abs\_int}, Frama-C/WP cannot establish that the precondition of \texttt{abs\_int} is satisfied or, in other words, that \( x > \text{INT\_MIN} \) holds.

```c
#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 \texttt{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 \texttt{abs\_int} is correct.

Note that this assertion is a static one, that is, it is an AC SL annotation that resides inside a comment and does not affect the execution of the code in Listing 2.10. Also note that unlike
### listing 2.10. A more complex example of modular verification

```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;
    // ...
}
```

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

### 2.6 Dealing with side effects

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

```c
#include <limits.h>
extern int a;

/*@ requires x > INT_MIN; */
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.
#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

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

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

    To show that all software modules, elements and subsystems interact correctly to perform their intended function and do not perform unintended functions (see also [2, §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.

//@ 
  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 Chapter 5 on the lowest level.

On the data packet level, a total of 47 different packets are defined as C struct. We exemplify our discussion on the alphabetically first packet, AdhesionFactor (Section 3.2), and give some comments on considerations with respect to other packets (Section 3.3).

In order to cope with the similarity of specification, implementation, and verification tasks for all packets, we have chosen to automatically generate formal specifications and implementations for encoding and decoding data packets from chapter 7 of the ETCS Subset 026 system requirements description.

3.1 Overview and classification of data packets

Tables 3.1, 3.2, 3.3, and 3.4 list the packets from ETCS requirements specification “Subset 026”. While many packets only contain a fixed number of elements, there are also a few that contain a dynamic number of elements (indicated by the presence of a field N_ITER) and optional values. We use the following terminology:

- A packet that does not contain an element N_ITER is referred to as static packet.
- All other packets are referred to as dynamic packets.

There are 29 static packets and 19 dynamic packets. Seven of the 29 static packets contain optional values.
<table>
<thead>
<tr>
<th>PacketID</th>
<th>Packet name</th>
<th>N_ITER</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>National Values</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>5</td>
<td>Linking</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>12</td>
<td>Level 1 Movement Authority</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>Level 2/3 Movement Authority</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>Repositioning Information</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>Gradient Profile</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>27</td>
<td>International Static Speed Profile</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>39</td>
<td>Track Condition Change of traction power</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>41</td>
<td>Level Transition Order</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>42</td>
<td>Session Management</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>Data used by applications outside the ERTMS/ETCS system</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>45</td>
<td>Radio Network registration</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>Conditional Level Transition Order</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>49</td>
<td>List of balises for SH Area</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>51</td>
<td>Axle load Speed Profile</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>57</td>
<td>Movement Authority Request Parameters</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>58</td>
<td>Position Report Parameters</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>63</td>
<td>List of Balises in SR Authority</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>65</td>
<td>Temporary Speed Restriction</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>66</td>
<td>Temporary Speed Restriction Revocation</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>67</td>
<td>Track Condition Big Metal Masses</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>68</td>
<td>Track Condition</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>70</td>
<td>Route Suitability Data</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>71</td>
<td>Adhesion Factor</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>72</td>
<td>Packet for sending plain text messages</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>76</td>
<td>Packet for sending fixed text messages</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>79</td>
<td>Geographical Position Information</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>80</td>
<td>Mode profile</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>90</td>
<td>Track Ahead Free up to level 2/3 transition location</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>131</td>
<td>RBC transition order</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 3.1.** TrackToTrain packets part 1
<table>
<thead>
<tr>
<th>PacketID</th>
<th>Packet name</th>
<th>N_ITER</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>Danger for Shunting information</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>133</td>
<td>Radio in-fill area information</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>134</td>
<td>EOLM Packet</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>136</td>
<td>Infill location reference</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>137</td>
<td>Stop if in Staff Responsible</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>138</td>
<td>Reversing area information</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>139</td>
<td>Reversing supervision information</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>140</td>
<td>Train running number from RBC</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>141</td>
<td>Default Gradient for Temporary Speed Restriction</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>254</td>
<td>Default balise, loop or RIU information</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 3.2. TrackToTrain packets part 2**

<table>
<thead>
<tr>
<th>PacketID</th>
<th>Packet name</th>
<th>N_ITER</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Position Report</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>1</td>
<td>Position Report based on two balise groups</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>Onboard telephone numbers</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>Error Reporting</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>Level 2/3 transition information</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>Validated train data</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>44</td>
<td>Data used by applications outside the ERTMS/ETCS system</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 3.3. TrainToTrack packets**

<table>
<thead>
<tr>
<th>PacketID</th>
<th>Packet name</th>
<th>N_ITER</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>255</td>
<td>End of information</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 3.4. Both-Ways packets**

This work is licensed under the "openETCS Open License Terms" (oOLT).
3.2 **Formal specification of AdhesionFactor**

In this section we describe in detail the formal specification of encoding and decoding operations of the packet `AdhesionFactor`, which is a typical static packet.

### 3.2.1 AdhesionFactor in ETCS

ETCS Subset 026 defines the package *adhesion factor* (packet 71) as shown in Table 3.5.

<table>
<thead>
<tr>
<th>variable name</th>
<th>number of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>NID_PACKET</td>
<td>8</td>
</tr>
<tr>
<td>Q_DIR</td>
<td>2</td>
</tr>
<tr>
<td>L_PACKET</td>
<td>13</td>
</tr>
<tr>
<td>Q_SCALE</td>
<td>2</td>
</tr>
<tr>
<td>D_ADHESION</td>
<td>15</td>
</tr>
<tr>
<td>L_ADHESION</td>
<td>15</td>
</tr>
<tr>
<td>M_ADHESION</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.5. Packet `AdhesionFactor` as defined by ETCS

### 3.2.2 The type `AdhesionFactor`

Listing 3.1 shows the definition of type `AdhesionFactor` as it is generated from the ETCS specification shown in Section 3.2.1.

```c
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. Definition of the type `AdhesionFactor`
3.2.3 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.

```acsll
/*@ 
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 an object of type AdhesionFactor.

```acsll
/*@ 
logic integer Invariant(L)(AdhesionFactor* p) = Invariant(q_dir,L)(p) &
Invariant(q_scale,L)(p) &
Invariant(d_adhesion,L)(p) &
Invariant(l_adhesion,L)(p) &
Invariant(m_adhesion,L)(p);
*/
```

Listing 3.3. Definition of the Invariant predicate for 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);
*j/

Listing 3.3. Definition of the Invariant predicate for AdhesionFactor

Listing 3.4 shows the definition of the UpperBitsNotSet predicate for 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);
*j/

Listing 3.4. Definition of the UpperBitsNotSet predicate for AdhesionFactor

The predicate UpperBitsNotSet(AdhesionFactor*) holds if and only if the values of all members of AdhesionFactor fit into their assigned numbers of bits. The predicate is defined as the conjunction of the overloaded UpperBitsNotSet predicate which is explained in Section 5.2 for all members of 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);
*j/

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.

This work is licensed under the “openETCS Open License Terms” (oOLT).
/**
 * 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

### 3.2.4 Formal specification of AdhesionFactor\_UpperBitsNotSet

Listing 3.7 shows the contract of the UpperBitsNotSet function for AdhesionFactor.

```c
/*@ 
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

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 3.2.3. The contract also includes 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 3.2.3. With the assigns \nothing clause the contract furthermore specifies that this function has no side effects.
3.2.5 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.

/*@
   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_DecodeBit(AdhesionFactor* p, Bitstream* stream);

Listing 3.8. Contract for DecodeBit function of AdhesionFactor

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 behaviors’ 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.

- The property valid_stream requires that the predicate Readable(stream) is satisfied (see Section [4.1]).
• The property `stream_invariant` is only met if the `Invariant` predicate is true. The predicate `Invariant(Bitstream*, integer)` is described in Section 4.1.

• The property `valid_package` requires that `p` is a valid pointer for read and write operations.

• The property `separation` requires that `stream` and `p` do not overlap in the memory. The `Separated` predicate was introduced in Section 3.2.3.

The behavior `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.

The behavior `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 in particular 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 was 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 Section 4.1.
3.2.6 Formal specification of AdhesionFactor_EncodeBit

Listing 3.9 shows the contract for the EncodeBit function for AdhesionFactor.

```c
/*@ requires valid_stream: Writable(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_EncodeBit(const AdhesionFactor* p, Bitstream* stream);
```

Listing 3.9. Contract for EncodeBit function of AdhesionFactor

The behavior of the function is described using the three disjoint behaviors normal_case, values_too_big and invalid_bit_sequence. The requirements valid_stream, stream_invariant, valid_package, invariant and separation are similar to those of the DecodeBit function’s contract for AdhesionFactor. The ones not examined in detail here do not differ from the ones in Section 3.2.5.
Like for the DecodeBit function for AdhesionFactor in Section 3.2.5 the assigns clauses in the contract body are the conjunction of the assigns clauses of the individual behaviors.

- **Property valid_stream** is only met if Writable(stream) applies. The predicate Writable(Bitstream*) requires that the stream is accessible for updates.

- **Property valid_package** requires *p to be valid pointer.

- **Property invariant** is only met if the Invariant predicate, which was described in Section 3.2.3 holds for p.

The behaviors of the EncodeBit contract describe one successful case and two error cases.

Behavior **normal_case** describes a successful encoding of the object *p into the bit stream. The assigns clauses specify that in this case both the bitpos of the stream and the fields of the bit stream are written. The increment clause describes the new value for bitpos. The ensures clauses left, middle and right state that only some bits of the bit stream are written. The updated bits and their relationship to the bits of the members of the object *p are described with the EqualBits predicate, which is described in Section 3.2.3. The Unchanged predicate specifies that the bits in the bit stream before the old stream->bitpos and the after the new stream->bitpos remain unchanged. Unchanged(Bitstream*, integer, integer) is defined in Section 4.1.

Behavior **values_too_big** describes the scenario in which the value of at least one member of *p is bigger than the specified bit size for that member of AdhesionFactor allows. The numbers of bits for the members of AdhesionFactor are specified in Section 3.2.1. The assigns clause states that this behavior of the function causes no side effects and the result clause ensures that the function will return the value -2. In contrast to normal_case, for this behavior it is assumed that the UpperBitsNotSet(p) predicate evaluates to false. The Normal(stream, MaxBitSize(p)) predicate returns true for both behaviors.

Finally, the behavior **invalid_bit_sequence** describes the function’s behavior if the bit stream is not long enough to write a complete AdhesionFactor object into. This behavior is distinguished from the other behaviors by the evaluation of the predicate Normal(stream, MaxBitSize(p)). Notice that the evaluation of UpperBitsNotSet(p) might be false, too. Like in the value_too_big behavior the function ends without encoding any bits into the stream. Therefore the assigns clause is \nothing. The result clause states that the function’s return value equals -1.
3.3 Formal specification of other packets

After examining the definition of the predicates for \(\text{AdhesionFactor}\) and the formal specifications of the functions \(\text{AdhesionFactor\_EncodeBit}\) and \(\text{AdhesionFactor\_DecodeBit}\) from Section 3.2, the predicates for other (static) packets and the specifications of the corresponding functions look very familiar.

In fact the only difference of the predicates lies in the different sets of sub predicates which are used and depends on the different sets of member variables for the different packets. The fact that ACSL predicates and logic functions can be overloaded for different types enables us to develop generic formal specifications for the \(\text{EncodeBit}\) and \(\text{DecodeBit}\) functions of all packets.
4 The bit stream layer

In this chapter, we describe the intermediate abstractions levels the packet level (Chapter 3) relies on. First, we discuss in Section 4.1 a level where operation arguments typically include a pointer to the C structure Bitstream, which encapsulates bitstream data and all related administration information (see Listing 4.1).

```c
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 4.1. Details for the Bitstream data structure

In this chapter, we present a level still working on bit sequences, but with an operation typically having one argument for every bitstream data or administration input. Chapter 5 finally presents lower level implementation details.

4.1 The Bitstream abstraction

The operations on packet data structures were implemented by operations on a `struct Bitstream*` argument. 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 4.2. It requires `stream`

- to point to a valid memory area (requirement property “valid”),
- to adhere to its data type invariant (property “invariant”), and
- not to be exhausted (property “normal”).
Listing 4.2. Reading from a bitstream

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 (postcondition property “pos”),
- the return value equals, bit by bit, the stream between the current bit position on entry and that on exit (property “changed”),
- in particular, all but the length least significant bit(s) of the return value are zero (property “upper”),
- stream’s total size remains unaffected (property “size”), and
- so do all of its content bits (property “unchanged”).

Figure 4.1. Bit coincidences required by EqualBits

---

3 Bit positions are counted differently in Frama-C and in the openETCS project. In this report, we preferably used the terms “least” and “most significant bit(s)” to designate a (range of) bit position(s) independent of the coordinate system.
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 4.1.

```plaintext
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

- Predicate Readable requires that a stream’s data area is complete accessible for read.
- Similarly, predicate 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 4.3 below, in particular Listing 4.7).
- In a similar way, predicate Normal and EqualBits is reduced to a lower-level predicate of the same name, respectively.
- 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.
- A clause 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 4.3.

---

4Frama-C allows for predicate overloading.
5 We tacitly assume that each stream has a multiple of 8 bits available.

---

This work is licensed under the “openETCS Open License Terms” (oOLT).
4.2 Auxiliary Bitstream functions

As a kind of constructor for type Bitstream, we provide the operation Bitstream_Init, shown with its contract in Listing 4.4

```c
/*@ 
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

Moreover, we provide a test for exhaustion of a Bitstream, shown in Listing 4.5

```c
/*@ 
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
4.3 Writing bit sequences

In this section, we describe the function `Bitstream_Write` that transfers the designated bits from a variable of type `uint64_t` into a bit stream (see Figure 4.2).

![Figure 4.2. Sketch of Bitstream_Write](image)

Listing 4.6 shows contract of the `Bitstream_Write` operation, and moreover exemplifies its implementation.

```c
/*@ 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
Most parts of the contract are quite similar to that of Bitstream_Read in Listing 4.2. 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 property “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.

The formal definitions of the used ACSL predicates are given in Listing 4.7. Again, the tacit assumption that the array contains sensible data up to 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 4.7. ACSL predicates used in bit sequence layer contracts

The implementation just employs the lower-level operation Bitwalker_Write to write the bits, and appropriately updates the stream’s bitpos. Listing 4.8 shows the contract, and the implementation, of the Bitwalker_Write operation.

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 subsequently destroyed by the increment of bitpos.
/**
  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 4.8. Writing a bit sequence
4.4 Reading bit sequences

In this section, we describe the function Bitstream_Read that transfers the designated bits from a bitstream into a variable of type uint64_t (see Figure 4.3).

![Figure 4.3. Sketch of Bitstream_Read](image)

The following peculiarities are observed when the former is compared to Bitwalker_Read’s contract in Listing 4.9.

```c
/*@ 
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 4.9. Reading a bit sequence
• We require that the value to be written fits into the specific length, i.e. all but its length least significant bits are zero (requirement property “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 property “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 property “left” and “right”, respectively.

• In the i\textsuperscript{th} iteration, the bits [bitpos...bitpos+i) agree with the least significant i bits of value—invariant property “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 throughout the whole loop execution, but is decreased in every iteration — loop variant clause. Therefore, the loop is guaranteed to terminate eventually.

The operations we have discussed here are 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 Chapter [5]
4.5 Verification of the Bitstream abstraction

Critics of the formal software verification approach often argue that verifying an operation against its formal specification results in little or no increase of trustworthiness when

- the specification, including all auxiliary definitions etc., is as complex as the operation’s implementation, or/and
- the specification essentially duplicates the implemented algorithm in a different (such as functional rather than imperative) language.

Both criteria may be seen to be met by our Bitwalker case study.

However, since the operations we dealt with essentially implement a communication protocol, there is a very simple “high-level” property that should be satisfied, viz. that a “send” operation is inverse to a “receive” operation. This property can be stated formally in a very brief and understandable way. It ensures, in a mathematical context, that both operations implement bijective mappings, that is, in an engineering sense, that the communication channel neither looses, nor subjoins information. In fact, we have achieved to formally prove this property.

More particularly, in our setting, we could show that the operations `Bitstream_Read` and `Bitstream_Write` are inverse to each other. To this end, we set up two fictitious C procedures realizing the composition of both operations in the two possible orders.

4.5.1 The verification function `Bitstream_WriteThenRead`

Figure [4.4](#) outlines the first scenario where a 64-bit value is written into a bit stream and read from there afterwards (see Figure [4.4](#)).

---

This work is licensed under the "openETCS Open License Terms" (oOLT).
Listing 4.10 shows the procedure for the scenario “use Bitstream_Write to write a value to a stream, then immediately read it back using Bitstream_Read”.

```c
uint64_t Bitstream_WriteThenRead(Bitstream* stream, uint32_t length, uint64_t v) {
    Bitstream_Write(stream, length, v);
    uint64_t result = Bitstream_Read(stream, length);
    return result;
}
```

The procedure’s body code is straightforward; after Bitstream_Write, we have to seek back to the original bit position, before calling Bitstream_Read”. We could show that the read value always equals the written one, provided

- the stream is accessible for both read and update (requirement property “valid”),
- it satisfies its type invariant (property “invariant”; cf. Listing 4.3 and 4.7),
- the stream’s current bit position is sufficiently small such that all value bits still fit into the stream (property “normal”), and
- the most significant value bits that are not written are all zero (property “upper”).

This ensures that the bitstream communication channel doesn’t loose information—every value we write into it can completely be restored.

This work is licensed under the “openETCS Open License Terms” (oOLT).
4.5.2 The verification function Bitstream_ReadThenWrite

Vice versa, we could also show that the channel doesn’t transmit more information than is needed to fulfill its task. Figure 4.4 outlines this second scenario where a 64-bit value is read from a bit stream and written into another bit stream afterwards (see Figure 4.5).

Figure 4.5. Outline of Bitstream_ReadThenWrite

Listing 4.11 shows the procedure for the scenario “use Bitstream_Read to read a value from a stream, then immediately write it back using Bitstream_Write”.

```
/*@ requires valid: Writeable(stream);
requires invariant: Invariant(stream, length);
requires normal: Normal(stream, length);
assigns stream->addr[0..stream->size-1];
assigns stream->bitpos;
ensures unchanged: Unchanged{Here,Old}(stream, 0, 8 * stream->size);
*/
void Bitstream_ReadThenWrite(Bitstream* stream, uint32_t length)
{
    //@ ghost uint32_t old_pos = stream->bitpos;
    uint64_t value = Bitstream_Read(stream, length);
    //@ assert equal: EqualBits(stream, old_pos, old_pos+length, value);

    stream->bitpos -= length;
    //@ assert stream->bitpos == old_pos;

    Bitstream_Write(stream, length, value);
    //@ assert unchanged: Unchanged{Here,Pre}(stream, old_pos, stream->bitpos);
}
```

Listing 4.11. Verifying the scenario “read, then write”
We were able to show that this leaves the whole stream unchanged, Provided the first three 
requirements from Bitstream_WriteThenRead are met. As an example for 
a channel transmitting redundant information, consider a bitstream implementation with 
Bitstream_Write storing each byte twice in succession and Bitstream_Read ignor-
ing every second byte. Such a stream doesn't meet our property, since, starting from a stream 
with non-agreeing adjacent bytes, there is no way to reproduce it by a “read, then write” scenario.
5 Low-level bitstream operations

In this chapter, we describe the implementation of the low-level bitstream operations. They were used to implement the bit sequence abstraction level, cf. Chapter 4. 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 5.1.1 for an array of uint8_t, in Subsection 5.1.2 for a single uint8_t, and in Subsection 5.1.3 for single uint64_t.

5.1 Reading and writing individual bits

5.1.1 Reading and writing 8 bit arrays

In this section, we discuss the operations for read and write of a single bit from/into a byte array.

The operation TestBit8Array(addr, size, pos) returns the pos\textsuperscript{th} bit within the array at addr of byte-size size. Its contract and its implementation is shown in Listing 5.1. See Listing 5.8 for the definition of the predicate Bit8Array. The array bits are counted starting with the most significant one of the first byte. A call to TestBit8(bytevalue, bitadr) returns the bitadr\textsuperscript{th} bit within bytevalue, this operation is discussed in Subsection 5.1.2 below.

```c
/*@ 
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 5.1. Reading a bit of an uint8_t array

Similarly, the operation SetBit8Array(addr, size, pos, flag) sets the pos\textsuperscript{th} bit within the array at addr of byte-size size to flag. Its contract is shown in Listing 5.2. It requires

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

This parameter isn’t actually used in the code, but merely in the contract.
The assigns clause allows the operation to change the contents of the array, but no other memory locations. On completion, the operation shall guarantee

- that the value of flag is actually stored at the designated bit position (postcondition property “middle”; the call Bit8Array() succeeds if, and only if, the pos\textsuperscript{th} bit within the byte array at addr is set, cf. Listing 5.8 in Section 5.2), and
- that all other bits remain unchanged (properties “left”, “right”).

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

```c
/*@ 
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 5.2. Writing a bit of an uint8\_t array

\footnote{Any non-zero flag value is treated like 1. This is ensured by the contract of the called operation SetBit8, cf. Subsection 5.1.2.}
5.1.2 Reading and writing 8 bit words

```c
/*@ 
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 5.3. Reading a bit of uint8_t

The operation `TestBit8(value, pos)` returns the `pos`th bit of `value`. Its contract is shown in Listing 5.3.

- The value of `pos` must not exceed 7 (requirement property "pre"),
- no memory may be modified (assigns), and
- the result is non-zero if, and only if, the specified bit is set (postcondition property "pos"; the call `Bit8(value, pos)` succeeds if, and only if, the `pos`th of the byte `value` is set, cf. Listing 5.8 in Section 5.2).

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.

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 5.4.

- Again, the value of `pos` mustn’t exceed 7 (requirement property “pre”),
- no memory may be modified (assigns clause),
- the return value coincides with `value`, except possibly at `pos` (postcondition properties “left” and “right”; a call `EqualBits8(x, y, first, last)` succeeds if, and only if, the `uint8_t` values `x` and `y` agree on all bits in range `[first...last]`, cf. also Listing 5.10 in Section 5.2), and
- `flag` is written to the appropriate bit of `value` (property “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) \iff (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 5.4. Writing a bit of uint8_t

5.1.3 Reading and writing 64 bit words

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 5.5 and 5.6 without repeating the comments given in Subsection 5.1.2 for the 8 bit version. See Listing 5.8 for the employed ACSL predicates.

/*@ 
requires pre: pos < 64;
assigns \nothing;
ensures set_bit: \result != 0 \iff 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 5.5. Reading a bit of uint64_t

Listing 5.6 shows the operation SetBit64. Note that it has a redundant postcondition, viz. property “upper”, which guarantees that the leading zeros in value are kept in the result, up to, but excluding position pos. This property was needed to enable the provers to verify code that uses SetBit64.
/*@
  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 5.6. 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 in the implementation of packet writing functions like AdhesionFactor_EncodeBit (see Section 3.2.6) to check that no non-zero bits from the packet structure (like struct AdhesionFactor) are ignored due to space limitations in the bitstream.

/*@
  requires pre: length <= 64;
  assigns \nothing;
  ensures not_set: \result <==> UpperBitsNotSet(value, length);
 */
int UpperBitsNotSet64(uint64_t value, uint32_t length);

Listing 5.7. Test that upper bits are not set
5.2 Formalization of bit operations in Frama-C

The definition of predicate `Bit8` is shown in Listing [5.8]. It relies on the Frama-C library predicate `BitTest`, performing a coordinate transformation to fit Frama-C’s notion of bit positions with the notion of ETCS.

```
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 5.8. Definition of bit test predicates

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 [5.9].

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

Listing 5.9. Definition of the low-level predicate `UpperBitsNotSet`

Listing [5.10] shows the predicate `EqualBits64` that was used in the 64-bit operations’ contracts. The call `EqualBits64(x,y,first,last)` succeeds if, and only if, the `uint64_t` values `x` and `y` agree on all bits in the range from `first` to `last`). The predicate `EqualBits8`, used in Subsection 5.1.2, is defined in similar way; its definition need not be shown here.

```
predicate EqualBits64(A)(uint64_t x, uint64_t y, integer first, integer last) = 
forall integer i; 64 - last <= i < 64 - first ==>(BitTest(x, i) <==> BitTest(y, i));
```

Listing 5.10. Definition of the low-level predicate `EqualBits64`

In order for the provers to find all low-level validation proofs, we needed to supply three redundant properties about `EqualBits64` and `UpperBitsNotSet`; they are shown in Listing [5.11]. Axiom `equal_bits64_0` states that two `uint64_t` values must be equal, if they agree on all 64 bit positions. Axiom `upper_bits_less_than` states that in a nonnegative number less than `2^n` all bits are zero, except for possibly the least significant `n` ones. The necessity of these extra axioms might indicate an incompleteness in Frama-C’s actual bit-operator theory; this is currently investigated. Axiom `equal_bits64_1` is just a (relaxed) rephrasing of the definition in Listing [5.10] using a different index scheme. It is necessary due to the provers’ weakness in applying index transformations.

---

This work is licensed under the "openETCS Open License Terms" (oOLT).
axiomatic BitProperties
{
    axiom equal_bits64_0:
        \forall \text{uint64_t } x, y; \text{EqualBits64}(x, y, 0, 64) \implies x == y;

    axiom equal_bits64_1:
        \forall \text{uint64_t } x, y, \text{integer } p, q;
        \forall \text{integer } k; p <= k < q
        \implies \text{let } j = q-1-k; \text{BitTest}(x, j) \iff \text{BitTest}(y, j))
        \implies \text{EqualBits64}(x, y, 64-(q-p), 64);

    axiom upper_bits_less_than:
        \forall \text{integer } x, n; x >= 0 \implies n >= 0 \implies
        \text{(UpperBitsNotSet}(x, n) \iff x < (1 \ll n));
}

Listing 5.11. ACSL axioms used in 64-bit contracts

Finally, for a nonnegative integer $v$, the predicate $\text{BitTest}(v, n)$ succeeds if, and only if, the $n^{th}$ bit is set in the binary representation of $v$, i.e. iff the truncating integer division of $v$ by $2^n$ yields an odd number. This predicate comes with the standard library of the Frama-C system, however, without any detailed documentation. Its declaration is shown in Listing 5.12.

predicate $\text{BitTest}(\text{integer } v, \text{integer } n)$;

Listing 5.12. The Frama-C library predicate $\text{BitTest}$
6  Formal verification with Frama-C/WP

In this chapter we introduce the formal verification tools used in this tutorial. We will afterwards present to what extent the functions from Chapters 3–5 could be deductively verified.

Within Frama-C, the WP plug-in \[3\] enables deductive verification of C programs that have been annotated with the ANSI/ISO C Specification Language (ACSL)\[4\]. The WP plug-in uses weakest precondition computations to generate proof obligations. To formally prove the ACSL properties, these proof obligations can be submitted to external automatic theorem provers or interactive proof assistants.

We used the WP plugin-in of Sodium release of Frama-C\[5\] together with the automatic theorem provers Alt-Ergo (version 0.99.1\[6\]), CVC4 (version 1.4) and Z3 (version 4.4.0\[7\]).

Here are the options of Frama-C that we used and that influence the number of generated proof obligations.

\[-wp\]
\[-wp=rte\]
\[-warn-\text{signed}=\text{downcast}\]
\[-warn-\text{signed}=\text{overflow}\]
\[-warn-\text{unsigned}=\text{overflow}\]
\[-wp=\text{no-bits}\]
\[-wp=\text{model \text{Typed+ref}}\]
\[-wp=\text{par 1}\]
\[-wp=\text{timeout 20}\]
\[-wp=\text{prover z3}\]
\[-wp=\text{prover cvc4}\]
\[-wp=\text{prover alt-ergo}\]

We list in the following tables the number of generated verification conditions (VC), as well as the percentage of proven verification conditions. The tables show that all verification conditions could be verified.

Please note that the number of proven verification conditions do not reflect on the quality/strength of the individual provers. The reason for that is that we “pipe” each verification condition sequentially through Qed, Z3, CVC4 and Alt-Ergo. If one prover succeeds, then the remaining provers are not called.

\[8\]See \url{http://frama-c.com/install-sodium-20150201.html}
\[9\]See \url{http://alt-ergo.lri.fr}
\[10\]For the use of CVC4 (see \url{http://cvc4.cs.nyu.edu/web}) and Z3 (see \url{http://research.microsoft.com/en-us/um/redmond/projects/z3}) we relied on version 0.86 of the Why3 platform for deductive program verification (see \url{http://why3.lri.fr}).
6.1 Verification of bit stream and lower-level bit operations

Table 6.1 shows the result of formal verification for the functions from Chapters 4 and 5. All verification conditions could be automatically verified.

<table>
<thead>
<tr>
<th>component</th>
<th>vcs</th>
<th>individual provers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>all</td>
<td>proven (%)</td>
</tr>
<tr>
<td>bit stream</td>
<td>58</td>
<td>58</td>
</tr>
<tr>
<td>bit stream (inverse)</td>
<td>58</td>
<td>58</td>
</tr>
<tr>
<td>lower-level bit ops</td>
<td>126</td>
<td>126</td>
</tr>
</tbody>
</table>

Table 6.1. Verification results for bit stream and lower-level bit operations

6.2 Verification of static ETCS packets

Tables 6.2, 6.3, 6.4 and 6.5 show the results of formal verification for static packets.

All static packets without optional values could be automatically verified. The formal specification and verification of static packets with optional values or dynamic packets could not be completely executed in the course of the project and remains to be done as future work.

<table>
<thead>
<tr>
<th>PacketID</th>
<th>VC</th>
<th>VCs</th>
<th>Individual Provers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All Proven (%)</td>
<td>Qed</td>
<td>Alt-Ergo</td>
</tr>
<tr>
<td>16</td>
<td>436</td>
<td>436</td>
<td>100</td>
</tr>
<tr>
<td>39</td>
<td>483</td>
<td>483</td>
<td>100</td>
</tr>
<tr>
<td>42</td>
<td>559</td>
<td>543</td>
<td>97</td>
</tr>
<tr>
<td>45</td>
<td>389</td>
<td>389</td>
<td>100</td>
</tr>
<tr>
<td>57</td>
<td>483</td>
<td>483</td>
<td>100</td>
</tr>
<tr>
<td>65</td>
<td>624</td>
<td>624</td>
<td>100</td>
</tr>
<tr>
<td>66</td>
<td>389</td>
<td>389</td>
<td>100</td>
</tr>
<tr>
<td>71</td>
<td>530</td>
<td>530</td>
<td>100</td>
</tr>
<tr>
<td>72</td>
<td>999</td>
<td>957</td>
<td>95</td>
</tr>
<tr>
<td>76</td>
<td>958</td>
<td>938</td>
<td>97</td>
</tr>
<tr>
<td>90</td>
<td>474</td>
<td>465</td>
<td>98</td>
</tr>
</tbody>
</table>

Table 6.2. Verification results for static track to train packets part 1

This work is licensed under the "openETCS Open License Terms" (oOLT).
<table>
<thead>
<tr>
<th>PacketID</th>
<th>VCs</th>
<th>Individual Provers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>Proven (%)</td>
</tr>
<tr>
<td>131</td>
<td>624</td>
<td>624</td>
</tr>
<tr>
<td>132</td>
<td>389</td>
<td>389</td>
</tr>
<tr>
<td>133</td>
<td>718</td>
<td>718</td>
</tr>
<tr>
<td>134</td>
<td>624</td>
<td>624</td>
</tr>
<tr>
<td>136</td>
<td>474</td>
<td>465</td>
</tr>
<tr>
<td>137</td>
<td>389</td>
<td>389</td>
</tr>
<tr>
<td>138</td>
<td>483</td>
<td>483</td>
</tr>
<tr>
<td>139</td>
<td>483</td>
<td>483</td>
</tr>
<tr>
<td>140</td>
<td>389</td>
<td>389</td>
</tr>
<tr>
<td>141</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td>254</td>
<td>342</td>
<td>342</td>
</tr>
</tbody>
</table>

Table 6.3. Verification results for static track to train packets part 2

<table>
<thead>
<tr>
<th>PacketID</th>
<th>VCs</th>
<th>Individual Provers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>Proven (%)</td>
</tr>
<tr>
<td>0</td>
<td>926</td>
<td>910</td>
</tr>
<tr>
<td>1</td>
<td>973</td>
<td>948</td>
</tr>
<tr>
<td>4</td>
<td>342</td>
<td>342</td>
</tr>
<tr>
<td>9</td>
<td>342</td>
<td>342</td>
</tr>
<tr>
<td>44</td>
<td>389</td>
<td>389</td>
</tr>
</tbody>
</table>

Table 6.4. Verification results for static train to track packets

<table>
<thead>
<tr>
<th>PacketID</th>
<th>VCs</th>
<th>Individual Provers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>Proven (%)</td>
</tr>
<tr>
<td>255</td>
<td>239</td>
<td>239</td>
</tr>
</tbody>
</table>

Table 6.5. Verification results for both-ways packets

*This work is licensed under the "openETCS Open License Terms" (oOLT).*
7 Verification of integration

Figure 1.2 on Page 2 outlines the scope of code verification within OpenETCS. Verification activities, however, also concern integration of the C code with the SCADE model (see Section 7.1) and with the communication layers that underlies the bit stream abstractions (see Section 7.2).

7.1 Integration with SCADE model

The generated data packets from Chapter 3 are sent and received over the so-called bit stream layer (see Chapter 4). In order to exchange the packets with the OpenETCS application layer that is modeled in SCADE (see Figure 1.2) the so-called integer stream layer was defined. Due to the relative late emergence of this layer in the OpenETCS project this layer could not be formally be verified. Rather it was tested together with underlying communication layer (see Section 7.2).

7.2 Integration with the underlying communication layer

The ETCS packets and messages which are decoded represent the bit decoding and encoding layer as applied inside the BTM and EURORADIO module as described inside 11.

Concerning the ETCS air gap messages it represents the layer for bit encoding and decoding the ETCS content

- coded_EUROBALISE_input_telegram
- coded_EURORADIO_output_msg and
- coded_EURORADIO_input_msg

between the openETCS Kernel and the OpenETCS API (see Figure 7.1).

The scope of the ETCS data packets is aligned with the openETCS show case Amsterdam-Utrecht. Thus not the complete set of packets can be decoded. The supported packets are listed inside Section 3.1.

The OpenETCS decoder/encoder is referenced as “additional code components developed by other means” 12. It is generated from the requirement document of Subset-026-6 and SUBSET-026-7 to a XML model. This model was verified manually against the SUBSET-026. Although not having a certified code generator the verification is artifacts were verified after generation.

---

11 See SUBSET-026, Chapter 2, Figure 1.
12 See Chapter 3.5 of Deliverable 2.3.
Figure 7.1. Overview on integration interfaces.
8 Conclusion

The main results of this report are

- generation of specifications and contracts, i.e.,
  - generation of formal specifications from a formalization of ETCS packets
  - generation of C code from a formalization ETCS data packets
  - formal verification of a subset (static packets) of the generated code against the generated specifications
- formal verification of underlying bit operations
- integration with the underlying communication layer
- integration with code generated from SCADE model

8.1 Lessons learned

The formal verification provided important feedback regarding the applicability of Frama-C with respect to C code from the railway domain. The main technical challenge was to handle the automatic verification of low-level bit operations with the Frama-C/WP plugin. This was achieved by a properly defined layer of elementary bit operation.

8.2 Frama-C issues

While working with Frama-C in the course of the OpenETCS project, Fraunhofer FOKUS identified several issues of Frama-C and reported them together with feature requests to the developers at CEA LIST. Table 8.1 lists some of the issues reported by Fraunhofer FOKUS.

---

13Frama-C’s bug tracking system (BTS) can be accessed via [https://bts.frama-c.com](https://bts.frama-c.com).

This work is licensed under the "openETCS Open License Terms" (oOLT).
Table 8.1. Selection of Frama-C issues identified within the OpenETCS project

<table>
<thead>
<tr>
<th>BTS identifier</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001638</td>
<td>assigns clause appears unprovable</td>
<td>acknowledged</td>
</tr>
<tr>
<td>0001685</td>
<td>Axiomatic is recompiled when using several processes</td>
<td>assigned</td>
</tr>
<tr>
<td>0001687</td>
<td>Frama-C GUI discards candidate Coq proof</td>
<td>assigned</td>
</tr>
<tr>
<td>0001693</td>
<td>Generate footprint from reads clauses of logic declarations</td>
<td>assigned</td>
</tr>
<tr>
<td>0001694</td>
<td>Generate proof obligation for drivers when driver instantiate a logic acsl declaration</td>
<td>assigned</td>
</tr>
<tr>
<td>0001699</td>
<td>Develop strategies to efficiently run WP with different ATP and Coq</td>
<td>assigned</td>
</tr>
<tr>
<td>0001761</td>
<td>Check that all occurrences of *p in assigns are guarded by a \valid(p) in requires</td>
<td>resolved</td>
</tr>
<tr>
<td>0001771</td>
<td>quality of PDF files</td>
<td>resolved</td>
</tr>
<tr>
<td>0002041</td>
<td>unable to use lemma separated_region</td>
<td>acknowledged</td>
</tr>
<tr>
<td>0002040</td>
<td>assumes clause and labels</td>
<td>resolved</td>
</tr>
<tr>
<td>0002098</td>
<td>overloading of predicate fails</td>
<td>resolved</td>
</tr>
<tr>
<td>0002100</td>
<td>readability of Coq(?) names</td>
<td>acknowledged</td>
</tr>
<tr>
<td>0002161</td>
<td>redefinition of <strong>STDC_VERSION</strong></td>
<td>resolved</td>
</tr>
</tbody>
</table>

We wish to emphasize the importance of identifying, documenting, and fixing of problems related to verification tools because these activities helps to document appropriate instructions or constraints on the use of the tools [1 § 6.7.4.3].

8.3 Future work

With respect to ETCS and Frama-C, further development is necessary in order to

- improve the specification and verification of dynamic packets and
- better document verification techniques for bit operations in Frama-C
References


