TECHNISCHE UNIVERSITÄT DRESDEN

"Friedrich List" Faculty of Transport and Traffic Sciences Chair of Transportation Systems Engineering

term paper

SAFE SOFTWARE DEVELOPMENT FOR A VIDEO-BASED TRAIN DETECTION IN ACCORDANCE WITH EN 50128*

Moritz Dorka
born on March 18th, 1988 in Kirchen (Sieg)
matr. no. 3472533, moritz.dorka@mailbox.tu-dresden.de

Examined by:
Prof. Dr. rer. nat. Jörg Schütte and Dr.-Ing. Sven Scholz
Supervised by:
Dr.-Ing. Sven Scholz and Dipl.-Ing. Nils Kawan
Submitted on July 24th, 2013

*German title in conformance with the "Richtlinie für die Anfertigung der Studienarbeit":

Untersuchung von Softwareimplementierungen für sicherheitskritische Anwendungsfälle

Dresden, July 24th, 2013 signature of the student

Bibliographic Data

DORKA, Moritz:
Safe software development for a video-based train detection in accordance with EN 50128
Studienarbeit Dresden 2013, 59 pages, 16 Figures, 4 Tables, softbound

ABSTRACT

This paper intends to give an overview of selected parts of the software development process for safety-relevant applications using the example of a video-based train detection. An IP-camera and an external image processing computer were equipped with a custom-built, distributed software system. Written in Ada and C, the system parts communicate via a dedicated UDP-based protocol. Both programs were subject to intense analysis according to measures laid down in the EN 50128 standard specifically targeted at software for railway control and protection systems.
Preceding each section, a structure resembling the standard document with references to the discussed measures allows for easy comparison with the original requirements of EN 50128.

In summary, the techniques have proven to be very suitable for practical safe software development in all but very few edge-cases. However, the highly abstract descriptive level of the standard requires the staff involved to accept an enormous personal responsibility throughout the entire development process. The specific measures carried out for this project may therefore not be equally applicable elsewhere.

Diese Studienarbeit gibt einen Überblick über ausgewählte Teile des Softwareentwicklungsprozesses für sicherheitsrelevante Applikationen am Beispiel eines videobasierten Zugerkennungssystems. Eine IP-Kamera und ein externer Bildverarbeitungscomputer wurden dazu mit einer speziell entworfenen, verteilten Software ausgestattet. Die in Ada und C geschriebenen Teile kommunizieren dabei über ein dediziertes, UDP-basiertes Netzwerkprotokoll. Beide Programme wurden intensiv anhand verschiedener Techniken analysiert, die in der Norm EN 50128 festgelegt sind, welche sich speziell an Software für Eisenbahnsteuerungs- und überwachungssysteme richtet.
Eine an der Norm orientierte Struktur mit Verweisen auf die diskutierten Techniken zu Beginn eines jeden Abschnitts erlaubt einen schnellen Vergleich mit den originalen Anforderungen des Normtexts.

Zusammenfassend haben sich die Techniken bis auf wenige Ausnahmen als sehr geeignet für die praktische Entwicklung von sicherer Software erwiesen. Allerdings entbindet die Norm durch ihre teils sehr abstrakten Anforderungen das am Projekt beteiligte Personal in keinster Weise von seiner individuellen Verantwortung. Entsprechend sind die hier vorgestellten Techniken für andere Projekte nicht ohne Anpassungen zu übernehmen.

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Verkehrswissenschaften „Friedrich List“

Themenblatt zur Studienarbeit*)

von Herrn cand. Ing. *Moritz Dorka*

Thema (Aufgabenstellung siehe Anlage):

Untersuchung von Softwareimplementierungen für sicherheitskritische Anwendungsfälle

Institut für: Bahnsysteme und Öffentlicher Verkehr, Professur für Verkehrssystemtechnik

1. Prüfer: Prof. Dr. Jörg Schütte

2. Prüfer / Beisitzer: Dr.-Ing. Sven Scholz

Zur Anfertigung der Studienarbeit wurde eine dreiseitige Vereinbarung (TUD, Student, Dritter) abgeschlossen: ~~Ja~~ / Nein (zutreffendes unterstreichen, nicht zutreffendes streichen)

Dresden, den 18. MRZ. 2013

Unterschrift des Prüfers
Prof. Dr. rer. nat. J. Schütte
Leiter der Professur

Ausgabetag: 24.04.2013

Abgabetermin: 24.07.2013

Bestätigung durch die Fakultät: .....

Abgabetag: ..... Bestätigung durch die Fakultät: .....

Bestätigung durch die Fakultät für eine genehmigte Verlängerung der Bearbeitungszeit: .....

Hiermit bestätige ich den Empfang der Aufgabenstellung für meine Studienarbeit und erkenne die Festlegungen der Richtlinie für die Anfertigung der Studienarbeit – insbesondere den Punkt 11 - an:

Dresden, den 24.4.2013

Unterschrift des Studierenden

*) siehe Diplomprüfungsordnung §§ 20 bis 24 sowie Studiendokumente 4., Punkt 2 der Regelung für die Ausgabe und Registratur der Studienarbeiten und Diplomarbeiten

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Verkehrswissenschaften „Friedrich List“
Institut für Bahnsysteme und Öffentlicher Verkehr, Professur Verkehrssystemtechnik

Aufgabenstellung zur Studienarbeit

für

Herrn cand. Ing. Moritz Dorka

zum Thema

Untersuchung von Softwareimplementierungen für sicherheitskritische Anwendungsfälle

An Software für sicherheitskritische Anwendungen, wie sie beispielsweise im Schienenverkehr vorkommen, werden hohe Anforderungen hinsichtlich der Qualität und Fehlerfreiheit gestellt. Dabei sollen üblicherweise systematische Fehler durch eine Vielzahl qualitätssichernder Design-, Codierung- und Prüfmaßnahmen auf ein absolutes Mindestmaß reduziert werden.

Anhand eines einfachen Anwendungsbeispiels zur automatisierten Detektion von Zügen in Videobildern soll ein Konzept für eine derartige Software erarbeitet und implementiert werden. Dabei sollen sicherheitsrelevante Anforderungen, wie sie beispielweise für Eisenbahnen-Software nach EN50128 gelten, beachtet werden.

Zunächst ist dazu ein geeigneter Detektionsalgorithmus zur Erkennung von Zügen in Videobildern zu entwerfern. Dieser Algorithmus ist so zu strukturieren, dass er als eigenständiges SW-Modul verwendet werden kann. Im nächsten Schritt soll untersucht werden, in welcher Form sich dieser Algorithmus implementieren lässt, wobei die Anforderungen an sicherheitskritische Softwaresysteme zu beachten sind. Es sollen geeignete Vorschläge für die Implementierung erarbeitet und in Form eines Katalogs von Anweisungen und Vorgaben zusammengestellt werden. Abschließend ist experimentell die notwendige Leistungsfähigkeit einer geeigneten Hardwareplattform zu prüfen. Dazu soll der implementierte Algorithmus direkt auf einer Kamera mit embedded Prozessor getestet werden.

Betreuer: Dr.-Ing. Sven Scholz,
Dipl.-Ing. Nils Kawan

Prof. Dr. Jörg Schütte

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Verkehrswissenschaften „Friedrich List“

Vorsitzender des Gemeinsamen Prüfungsausschusses für Verkehrsingenieurwesen und Bahnsystemingenieurwesen

Technische Universität Dresden, 01062 Dresden

Prof. Dr.-Ing.
Jochen Trinckauf

Telefon: 0351 463-36538
Telefax: 0351 463-36644
E-Mail: Jochen.Trinckauf@tu-dresden.de

Sekretariat: Eva Günther
Telefon: 0351 463-36697
Telefax: 0351 463-36644
E-Mail: Eva.Guenther@tu-dresden.de
Datum: 25.03.2013

Herrn
Moritz Dorka

c/o Prof. Dr. rer. nat. Jörg Schütte

Sehr geehrter Herr Dorka,

dem Antrag auf Verfassung der Studienarbeit in englischer Sprache vom 18. März 2013 wird unter folgenden Bedingungen stattgegeben:

1. Die Prüfer müssen ihr Einverständnis dahingehend erklären, dass sie die in der Fremdsprache angefertigte Studienarbeit zu bewerten gewillt sind. Die Gutachten sind auf jeden Fall in deutscher Sprache zu verfassen. Diese Einverständniserklärung ist dem Prüfungsausschuss in schriftlicher Form vorzulegen oder auf andere Weise glaubhaft zu machen.

2. Das Kolloquium zur Studienarbeit hat vorzugsweise in deutscher Sprache stattzufinden. Sollte sich das als nicht zweckmäßig erweisen, dann ist auf jeden Fall sicher zu stellen, dass in dem in der betreffenden Fremdsprache abgehaltenen Kolloquium auch Anfragen in deutscher Sprache gestellt und beantwortet werden können.

   Hinweis:
   Die Einverständniserklärung gemäß Ziffer 1 liegt bereits vor.

Mit freundlichen Grüßen

Prof. Dr.-Ing. Jochen Trinckauf

Postadresse (Briefe)
Technische Universität Dresden
01062 Dresden

Postadresse (Pakete u.ä.)
Technische Universität Dresden
Helmholtzstraße 10
01069 Dresden

Besucheradresse
Gerhart-Potthoff-Bau
Zimmer 257/258
Hettnerstr. 3
01067 Dresden

Internet
http://www.vkw.tu-dresden.de

Sekretariat
08:00 bis 12:00

THESES

1. Software exposed to high demands toward quality and freedom from defects is automatically regarded "safety relevant".
2. Software which is thoroughly examined by processes outlined in a standard document can be regarded "safe" according to that standard.
3. By following the standard systematic faults in software cannot be eradicated, but only minimized.
4. Train detection in software according to EN 50128 is not a simple use-case.
5. On his own, a single involved person is unable to give a cohesive set of requirements and guidelines compliant with EN 50128 which, once implemented, will turn an "ordinary software" into a "safe software".
6. Transferring the imagedata to a dedicated external computer and process it there is a superior approach compared to conducting this task on the camera itself.
7. A train detection algorithm may very well be parallelized for increased performance.
8. The embedded processor of an IP-camera is capable of serving uncompressed camera images via a network interface at frame rates sufficient for real-time train detections.

CONTENTS

1 INTRODUCTION

During the last decades computers have more and more become an essential part of our everyday life. *Ubiquitous computing* is the buzzword here, an umbrella term for the exponentially increasing cellphone hype and the urge to literally smarten up each and every toaster with some embedded chip [Tec05]. However, applications of computer technology can also be found in areas which perhaps will not come to mind as easily. One such field may be video-based train detection, the topic of this paper. But the control of nuclear power plants, army combat missiles, implanted pacemakers, steering-systems on commercial airplanes or high-frequency trading used in the financial sector fall into this category as well. What they all have in common is a high demand of *functional safety* (section 1.4) while often also requiring computations to finish within a given time, a task usually referred to as *real-time computing* (section 1.3).
The official definition of *functional safety* is rather complicated [Int10, part 4, page 23] but essentially it boils down to *safety*, which itself is defined as the absence of a fatal risk, in the context of a particular equipment which is to be controlled (so-called *EUC*).
Since this aim cannot be achieved easily, various standards have been created to cover not only the actual software development process for the controlled equipment but also the entire lifecycle of the overall system in which this particular equipment and hence also its supervising software may be integrated. Among the most notable representatives of those norms in the transport sector are DO-178B for aviation applications [RTC92] and the descendants of IEC 61508 [Int10], namely ISO 26262 [Int11b] for road vehicles and IEC 62279, which is more commonly known over here under the title of its European derivative EN 50128 [CEN11], for software applications in the field of railways.

This paper intends to give a use case for such a safety-relevant software system performing under real-time constraints for a railway application. The implications of the relevant norm EN 50128 on the development process are outlined.

1.1 Motivation

For surveillance purposes the interior of railway vehicles and the surrounding trackside infrastructure are nowadays often equipped with numerous CCTV cameras. The images obtained from those cameras are not only of interest to human operators, but may also be used as an input to automatic processing algorithms. Their aim is to quickly and reliably extract features from the image data, quantify them in some way and forward the result to other downstream algorithms.
In the concrete case of this paper an automatic train detector is to be developed, which can report back the presence of a train and its position in a track section on the basis of a single video image captured by such a CCTV camera. The focus is hereby placed on the architecture of software suitable for this task. The actual image analysis, however, is not a central part of this work and therefore only a rudimentary implementation has been elaborated so far (see section 2.4).

As the train position only provides relevant information when available within a given time, the resulting software must be capable of fulfilling real-time constraints (see section 1.3). This is

Figure 1: A typical camera image in different stages

why, among other factors, which will be discussed later, Ada was chosen for a major part of this project. Although merely a niche programming language when compared to C and C++, Ada offers extensive build-in concurrent- and real-time execution support, thus lending itself for this research issue.

1.2 Description of the problem

A camera (section 2.1) is mounted above the track in a station setting. The track itself may be shaped arbitrarily (curved) but its predominant direction is parallel to the Y-axis, so that the head of a train is clearly distinguishable on the camera images. The direction of movement on the rail, toward the camera or away from it, is unknown. However, the former is assumed to be the usual case. Image 1 on the left shows a typical picture obtained from such a camera.
The software analysis of this image is now supposed to identify the head of the train. The result should be the Y-pixel at which the train was detected (red line on image 1, middle). Since it can be assumed trains always move on rails, only the part of the image which contains the track, a so-called *Region of Interest*, actually needs to be forwarded to the detection algorithm (image 1, right).
The analysis itself is to be conducted on a per-picture basis. So the position of a train has to be computed by only a single picture as the input. Making it very easy to detect movements by differencing two consecutive images is not desirable because it would fail if the speed of the train became very slow, which is a situation not uncommon at the railway stations where the camera is to be installed. Differencing non-consecutive images is also impractical because n vs. n-m ($m > 1$) comparisons (i.e. differencing the current image with one from the more distant past) would only intensify the speed problem – at least if the framerate remained unchanged. A comparison with an "empty track"-image (i.e. one without a train present) is theoretically feasible but would mean quite some effort to constantly update this image at the right moment in time, so that changes of lighting, which would otherwise falsify the differentiation, are reflected correctly. Moreover, the edge-case of a non-moving train is not detectable by any of the differencing approaches.

1.3 Real-time constraints

Video images depict the outside world, which, of course, is not discrete in its nature but rather completely analogue. Hence, if a close resemblance of that world is desired a high sampling rate equalling many *frames per second* (fps) is inevitable. Each frame consists of numerous data, so the software handling these frames has to be fast enough, while transferring as well as while analyzing them and while printing the result subsequently (called $fps_{processed}$ in the formula below), in order to cope with the given rate of frames to be processed (fps_{min}). So *real-time* in this sense has nothing to do with "becoming as fast as nature" but rather with fulfilling a given, man-made, boundary condition:

$$fps_{processed} \geq fps_{min}.$$

Whether or not this inequation holds depends on many factors and essentially the entire system must be analyzed to receive a valuable result[1]. Furthermore, some of these factors are by definition invariant (e.g. the make of the camera and thus its onboard chip and processing speed) and only a few can be altered in order to eventually obtain a sufficient result – which is not necessarily the fastest in terms of speed as will be seen in section 1.4.

The only hard constraint given by the supervisors was 20 ms for the image-analysis part of the software. This means $f_{min} = 50\ fps$, which can easily be achieved on a modern PC. Tests conducted with `gprof` on a fairly recent 2.5 GHz Intel Core i5 yielded almost unmeasurable 0.62 ms for an average run of the corresponding function (`detect_train_one_rail()` in *Adaimageprocessor.Image.Analyze*) under worst-case circumstances (no train present). But taking into account the rest of the system, particularly the image acquisition process (copying the image from the internal memory on the camera, transferring it to the analyzing unit and similar tasks), the processing rate may decrease dramatically.
Unfortunately, the final system which the software is eventually to become part of is not yet fully defined. For instance, the camera images used so far were all non-live, static pictures with a much smaller resolution than the actual camera output (see section 2.1). Neither is any knowledge available about the network link of the camera, through which the pictures have to be transferred. Nor do any specifications exist regarding the computer responsible for the analysis of the images. All of these factors greatly influence the processing speed and thus the overall image refresh rate achievable. Therefore, a general answer on whether the inequation printed above can be satisfied or not is impossible to give here. All that can be said at this point is: The software has been designed in a way not to become the show-stopper.

1.4 Safety requirements

The position of a train on an image is defined as safety-relevant information. If for example this image describing the area to surveil was used as an input for some downstream obstacle detection system, a correct value would become eminently crucial. Otherwise, the obstacle detector

[1] So a little lookahead onto the upcoming sections is unavoidable here.

may identify the train itself as an obstacle, which is certainly not the desired behavior.

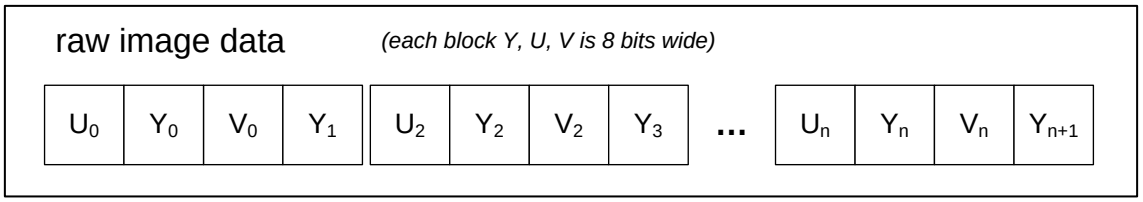
In order to achieve a commonly recognized level of safety, it is the aim of this paper to closely obey the guidelines of EN 50128 [CEN11]. The necessary Safety Integrity Level (SIL) has been identified as SIL 1 by the supervisors of this paper [SK12, p. 17].
As can be seen from the explanations in section 3, safety usually comes at the cost of a higher running time resulting from the overhead associated with error handling and similar safety-related measures, all of which contradict the aim of real-time performance as stated in section 1.3. In addition to that, higher SI-Levels usually mean higher monetary costs as well due to higher documentation, validation and verification demands by EN 50128. It should therefore be obvious to use an SI-Level as low as reasonably possible while also keeping the software as simple as possible.

2 IMPLEMENTATION DETAILS

There are two fundamentally different ways to implement a detection algorithm as it is required to tackle the problem laid out in section 1.2. Either, this algorithm can be run directly on the camera featuring an embedded Linux OS that comes with a suitable cross-compiler (see section 2.1). Or, the current video-image has to be extracted from the camera first and then processed on a separate computer.
The first approach is very simple and would probably have worked out perfectly for the given problem. However, the camera-processor is comparably slow and would at some point become the bottleneck if additional algorithms were to run in parallel as well. Furthermore, the Ada programming language, which is preferable over C for safety-relevant tasks (see section 3.2.5), is not supported by the cross-compiler.
With the second approach the programmer is usually limited to the predefined interfaces to the outside world implemented in the products by their vendors. These have proven to be imperfect for the purpose of this work, mainly because of the impossibility to avoid preprocessing which involves potentially dangerous code, and due to the limitation to data-intensive color images (for details see section 2.1).
Hence, it was decided to follow a hybrid approach. The camera was equipped with a small "server-program" which does not do anything but serve video-images in a predefined way to the outside world. Along with that, a "client-program" was programed which obtains these images and eventually processes them on a separate computer. Not only does this approach give maximum flexibility, but it also imposes the highest workload on the developer. In particular, a protocol (see section 2.2) must be developed to communicate between the server (camera) and the client – a task which would otherwise be left to the vendor.

2.1 Camera type and output format

The camera used for this project is an IP-based surveillance camera of the type M3314-R manufactured by Axis Communications AB, Lund, Sweden [Axi13a]. The camera features an ARTPEC-



Y – luminance-component of the respective pixel (= the grayvalue)
U – chrominance-component of the respective pixel based on its blue-value
V – chrominance-component of the respective pixel based on its red-value

Figure 2: The data structure of the UYVY-Format according to [Wil11]

3 SoC with 32 kB of cache and 92 MB of RAM which can be targeted by gcc's crisv32-port. According to its BogoMips-count the processor plays roughly in the league of a Pentium 1, although such simplifying comparisons are very inaccurate [Dor06].
Since Cris-based processors are not used by any other companies than Axis, they form a rather exotic build target. Cross-Compilers are hence only available from Axis directly and they are limited to the support of C and C++ on a 32-bit Linux host. The more recent ARTPEC-4 based cameras are built around an MIPS processor with numerous independent compilers existing. Among other things they also feature support for the Ada programming language.
Due to the widescreen resolution of 1280 by 800 pixels, its ruggedized design and the lack of a weatherproof protection, the camera is mainly targeted at applications inside railway vehicles. However, versions with a weatherproof case exist as well, and the limitation to the impractical resolution can be overcome by simply tilting the camera's lens by 90 degrees and swapping the output pixels accordingly. Marketed by Axis as their unique "Corridor Format" [Axi13b], this outcome is much more suitable for the project.

Images are captured in color with UYVY being their raw binary-format. For easy access to the camera's images it features an extensive HTTP-based API which emits the images either as an H.264 or an MJPEG encoded stream. Still images as BMPs or JPEGs are also available but they are of little use for real-time applications because of the overhead associated with establishing a new HTTP-connection for each transfer and the low frame-rate that would result from this approach. All requests to this API have to be made via the 100 Mbit/s ethernet interface of the camera.
Since the train detection algorithm discussed in section 2.4 only requires the gray-value of the pixels as its input, it makes sense to leave out all color-information right from the beginning. This is only possible by circumventing the API and accessing the camera's imagestore directly. As direct access also minimizes the use of external, possibly dangerous code, for example Axis's RAPP-library [Axi10] which would otherwise be used to preprocess images, this is also the preferable approach from a safety-point of view.
Axis allows this direct access via an IMAGE_UNCOMPRESSED interface in its SDK, which is shipped along with the cross-compiler. Because the UYVY pixel format employs a color subsampling focused on the abilities of the human eye [Sch05], color information (which is split into U and V components, based on the blue- respectively red-value of the pixel) is only available for each second pixel in a row (Figure 2). Hence, filtering out the remaining Y-information (the gray-values) is not completely straightforward. A possible implementation is given in the code below:

Listing 1: Reading out imagedata via the native interface in C

```
1  #include "capture.h"
2
3  typedef struct {
4    int width;
5    int height;
6    char data[MAX_HEIGHT][MAX_WIDTH];
7  } imagedata;
8
9  void readcamera(imagedata *output) {
10   media_native *nat = capture_open_native (MAX_WIDTH, MAX_HEIGHT);
11   capture_start_native (nat);
12   unsigned char *data = capture_get_image_native (nat);
13   {
14     int row, column;
15     int outcolumn = 0;
16     int offset_x1 = 0;
17     int offset_y1 = 0;
18     int offset_x2 = (*output).width;
19     int offset_y2 = (*output).height;
20     for (row = offset_y1; row < offset_y2; row++)
21       {
22         for (column = offset_x1*2+1; column < offset_x2*2; column=column
               +2)
23           {
24             outcolumn = column >> 1;
25             (*output).data[row][outcolumn] = (unsigned char*)data[row * (*
                   output).width * 2 + column];
26           }
27       }
28   }
29   capture_close_native(nat);
30 }
```

This code implies a ROI (section 1.2) which extends over the entire image. But this can be altered very easily by changing the variables in lines 16–19 accordingly and turning them into parameters to the function. Furthermore, the "Corridor Format"-issue mentioned earlier is not yet dealt with, which could be accomplished by swapping row with outcolumn in the left part of the assignment in line 25[2].
All called functions prepended with capture_* are defined in *capture.h* (line 1) which is a part of the SDK provided by Axis.

At the moment the entire function is written in C because it would have to run directly on the camera and remains unimplemented in the server-program, mainly due to the lack of a suitable test-installation of the camera. It was therefore easier to feed the downstream image-processing with saved images from disk, instead. This is the job of `camera_ReadImage()` in *camera.c* of the server-software which would have to be replaced by the code-snippet printed above for a production system.
Except for an additional header, PGM-images [Bor97] read in by `camera_ReadImage()` consist only of binary pixel values. Those can very easily be transformed into something suitable for the data-component of the imagedata-struct of lines 3–7 in the code above.

[2] For black-and-white images this is unproblematic. For color-images it would make a difference because the UYVY-subsampling (Figure 2) is not influenced by the rotation.

2.2 Transfer Protocol

To link the camera with the image-processing computer, i.e. the server-program with the client-program, a suitable transmission protocol had to be developed. The requirements toward this protocol essentially boiled down to these two simple points:

1. Very fast transfer of the image with as little overhead as possible because of the limiting 100 Mbit/s connection (see section 2.1).
2. Support of a Region of Interest (i.e. passing parameters to the server, see section 1.2).

As the other protocols (e.g. the *Real-Time Transport Protocol* [JFCS03]) are much more complicated in this field and therefore would have required much higher development efforts, a proprietary approach was followed. The resulting protocol is located somewhere between layers 5 to 7 of the OSI-model and is based upon UDP. Unlike its much more heavyweight counterpart TCP, UDP does not guarantee successful data transmission and is therefore inherently insecure. However, it also comes without the transmission initialization overhead (confer with section 2.1) of TCP and was therefore favored to comply with the first requirement. The price paid for using UDP is a higher workload on the developer's side, as the disassembly of the data on the server and the sequentially correct assembly on the client in order to fit this data into packages not exceeding the MTU of the underlying IP-network must all be coded into the software. TCP, in contrast, would take care of all these points automatically. For this reason, TCP is often preferred over UDP for large file transfers which require guaranteed transmission [SFR03, p. 596] . However, a certain rate of erroneous transfers can be tolerated in a real-time application since not receiving data at all (which is detectable and can be handled according to the fail-safe principle) is better than using old ones[3]. Moreover, because of its simplicity UDP is a supporting factor for the minimization of external code usage.

The protocol is based on a simple request/reply principle involving two letter codes as identifiers for each operation to ensure indempotence, possibly accompanied by parameters to fulfill requirement number two of the above list. For the actual data transfer only sequential numbers are used, though. An in-depth discussion of this implementation may be found in section 3.1.5. As socket programming is not natively supported by Ada, a compiler-specific extension (namely the *GNAT.Sockets* package) had to be utilized. This creates a conflict with the portability requirement of section 3.2.4[4]. But alternative implementations are available as well [sec13].

[3] In fact the client takes care of this via the second innermost loop, present in all but the very last (number 5) stages of the imagetransfer (Figure 16).

[4] But this is still superior to the C approach since *portability* here only refers to using different compilers. In the case of C the socket-implementation fails to stay portable even on the operating system's level.

Figure 3: Helper program written in wxPython to manually collect slice data

2.3 Real-world constrains

In order to fulfill the *real-time constraints* mentioned in section 1.3 entirely unacademic *real-world constraints* must be overcome first. The most important one is the camera itself: After all, one main reason to dig into video-based train detection is the fact that cameras are usually already present at the site, and hence "pimping them up" with some automation is initially a nice and cheap add-on to improve the safety of the railway system in some way or another. Conversely, this means that neither mounting, viewing angle or overall lighting situation in which the camera is embedded, nor the camera-model itself (especially regarding its resolution and image-enhancement features) have usually been specifically selected to facilitate computer vision.

The second major hindrance is the track layout. As the fundamental idea of the detection algorithm (see section 2.4 below) is to identify the train on the basis of the absence of the two rails on the image, knowledge about their position is essential. Automatic track detection algorithms for this purpose are indeed available (see section 4). However, they usually imply restrictions on the layout of the tracks such as having to begin at Y_{max} (i.e. the lower edge of the image) which does not hold for the example layout in Figure 1 where the left rail ends quite a few pixels earlier.

To overcome this constraint and at the same time avoid developing another dedicated track detection algorithm, a pixel-based approach was chosen. Each pixel belonging to the left, respectively right rail is simply saved into a large data array of the client-program, which in the case of Figure 1 may be found in *Adaimageprocessor.Image.Trackdata*. As it is terribly time consuming and error-prone to collect this data by hand, a little helper program called "ImageTagger" was

written in wxPython[5]. It allows the user to easily generate groups of pixels, so-called *slices* (see section 2.4 below), by simply clicking through a scaled picture obtained from the camera and eventually saving the coordinates of the slice-pixels in a format understood by Ada. All neighboring red, respectively yellow pixels make up one *slice* here. The rail itself is headed vertically, slightly tilted to the right – just as in Figure 1.
This "ImageTagger" also equips each *slice* with metadata regarding a threshold (see section 2.4, point 2) and a so-called `Link_Slice_Other_Rail`. The latter is intended for situations where a significant curvature in the track layout is present and hence a lot more slices are physically distinguishable on the outer rail than on the inner. With the output module currently assuming an approximately straight track layout by simply emitting the Y-pixel of the train position, this field remains unused.

2.4 Train Detection Algorithm

As stated in the motivation, the detection algorithm used in the software, specifically the client-part written in Ada, is not very sophisticated. Neither is it very accurate regarding the position of the head of the train, nor is the algorithm designed to detect a train in all conceivable situations. The former is partly due to the lack of a precise definition as to where a train actually begins (is it the bumpers or rather the chassis?) and the latter can be viewed as a tribute to the limitation of the single-picture analysis already discussed in section 1.2.

Figure 4: Working direction of the detection algorithm

The fundamental idea behind the detection is to prove that the track is *not* free – if this is the case, a train must be in the picture. To do so the algorithm takes one rail, divides it into small slices and iterates over them from beginning to end. Each of these slices consists of an array of points which form a part of the rail orthogonal to its longitudinal direction. Figure 4 shows the working direction of the algorithm for the right rail. Obviously, the number of pixels a slice consists of may vary due to the three-dimensional nature of the images: In the very back, the absolute (pixel-)width of the rail is significantly lower than in the front. Memory-wise Ada handles this issue very efficiently via the so-called *ragged arrays* (refer to the documentation of *Adaimageprocessor.Image.Trackdata* for details). The helper program discussed in section 2.3 automatically writes the pixel coordinates in this ragged format.
Iterating over all these slices, the algorithm now calculates their average pixel-value and out of the last ten slices the moving average, which is essentially a simple smoothing function[6]. The outcome of these operations for the case of the right rail of the image shown in Figure 3 on the left is plotted in Figure 5. If the moving average pixel-value is lower than any of its predecessors, its value is saved for later comparison (step 1 in Figure 5). The iteration continues until the difference between the moving average of the current slice

[5]That is: Python plus the wxWidgets library for creating a graphical user interface.
[6]Computed for a slice *m* by taking the average of the average of the last *n* (a parameter set to 10 for this project) slices relative to slice *m*.

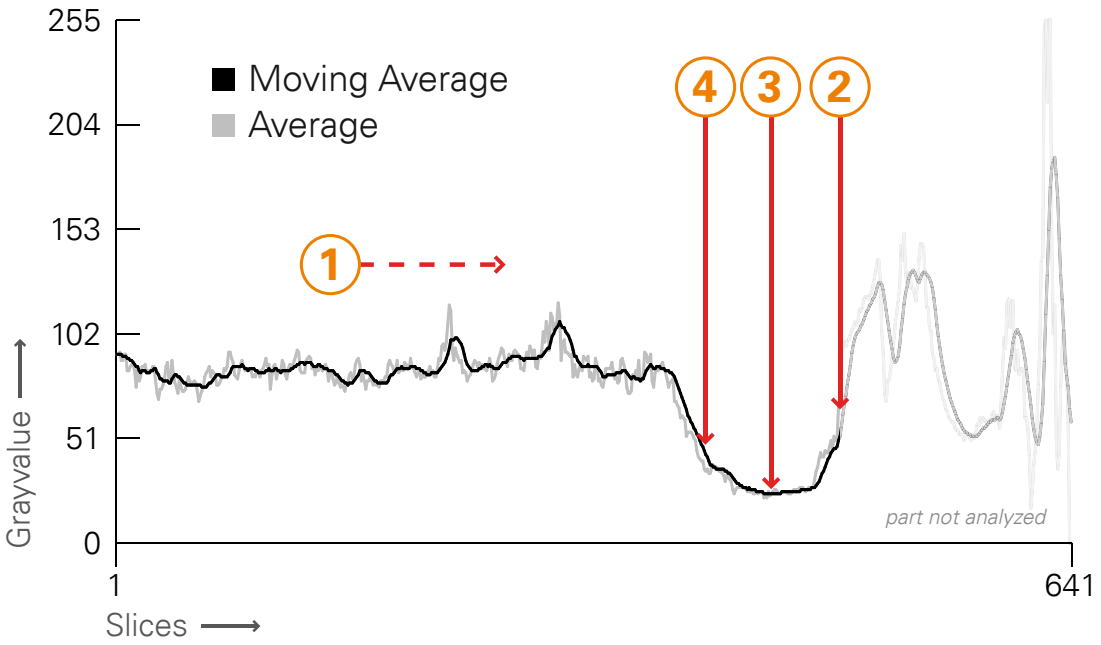


Figure 5: Plot of the average pixel values per slice for the case of the right rail in figure 1

and its predecessor exceeds a certain threshold value. For the example plot in Figure 5 this is the case at position 2. The algorithm now looks backwards until it finds a local minimum for the moving average (step 3). If this value and the absolute minimum from step 1 are roughly identical, it continues looking backwards and now checks if the slope of the moving average function stays within a certain range for a certain number of slices (step 4). This slope describes the shadow which is usually ahead of the train even in extreme lighting situations. If all those checks are successfully passed, the algorithm breaks and returns the Y-pixel of the slice at position 3 as the position of the train. In all other cases the iteration is repeated until the very last slice and eventually exits with "No train found". This is the worst case in terms of running-time, which was already discussed in section 1.3.
The same algorithm is now executed in parallel for the other rail as well and whichever of the two returns the lower value (suspects the train at a position further in the back of the image) wins. Only if both suggest that no train is present, the aforementioned message "No train found" is actually forwarded to the user and printed on the screen (see Figure 15).
A more programmatically-oriented discussion of this algorithm is given in section 3.2.2.

The downsides of this simple approach outlined above should be fairly obvious:

1. The train is expected to be rather bright, at least compared to its shadow on the rails.
2. There is no robustness against non-static shadows (moving passengers, other vehicles, . . .), static ones could be included via the individual thresholds of the slices in the Ada data structure which remain unused at the moment.
3. A train can only be detected as long as its head and the shadow in front of that head is visible. If the train occupies the entire image, the algorithm will not work. The case of a

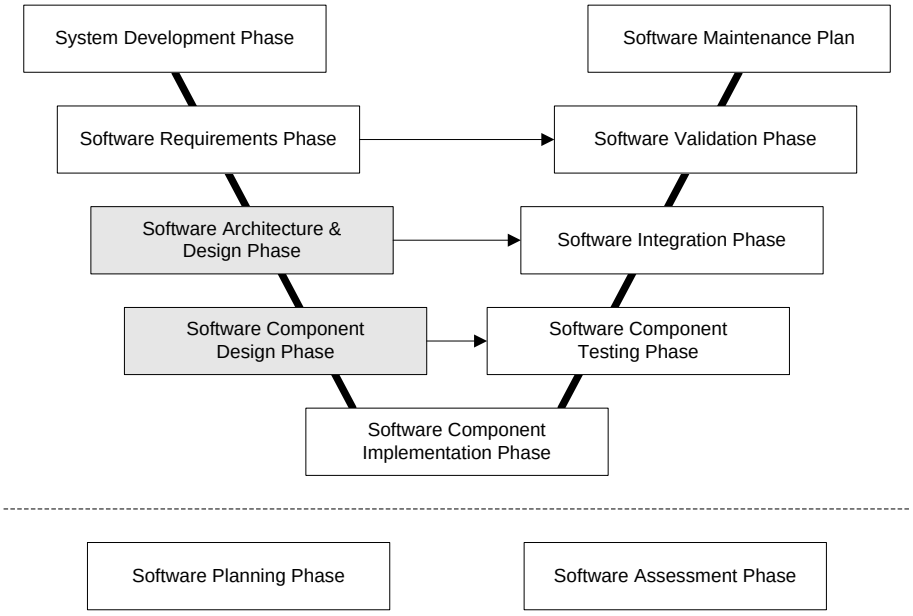


Figure 6: V-Model of the software development lifecycle according to [CEN11, p. 23]

train very close to the lower edge of the image (i.e. the shadow and possibly even the local minimum are not visible anymore) is not yet considered, either. However, the current implementation (`detect_train_one_rail()` in *Adaimageprocessor.Image.Analyze*) offers two exception-hooks to handle this situation.

The last point mentioned makes the algorithm in its current form unsuitable for production, since it would yield erroneous results in cases where the head of a train is not visible but the train is present, nevertheless. A straightforward solution to this dilemma would be to use image differencing at least for the time when the train occupies the entire image (during its halt at the platform). But this would violate the restriction regarding the exclusive use of single pictures for the analysis mentioned in section 1.2.

3 EN 50128 REQUIREMENTS

The European Standard 50128 has been around since 1995 and aims to harmonize the existing national procedures of approval for railway equipment (see section 3.2.5). This paper focuses on the part related to the actual software development of the standard which is oriented toward the so-called *V-Model*. Depicted in Figure 6 with the examined phases emphasized[7], this model describes the entire software lifecycle in a *waterfall*-like manner (i.e. one step after the other, confer with section 3.1.3). The central point (*Software Component Implementation Phase*) encompasses the actual coding of the software. Boxes on its left indicate preliminary checks whose outcomes may perhaps be checked by other downstream measures (horizontal arrows) on the right.

[7] The original version may be found in the standard here [CEN11, fig. 4, p. 23]

The core part of the *Software Architecture & Design Phase* and the *Software Component Design Phase* are tables of measures which must be carried out in order to reach compliance with a certain SI-Level for this phase[8]. The measures are divided into groups of *not recommended* (NR), *unspecified* (-) *recommended* (R), *highly recommended* (HR) and *mandatory* (M) ones. The last and the first group forbid, respectively require a certain measure to be undertaken [Bud11, p. 6]. The difference of the other two lies in the documentation overhead for justifications why they have not been considered [CEN11, p. 17]. Fortunately, each table comes with a set of *approved combinations* for each SIL which exempt from this rule and upon which the upcoming sections will be based.
The application of the individual techniques will be presented by examples, primarily taken from the client-program written in Ada, as this is considered to represent the integral safety-relevant part of the project. As it has been stated in section 1.4 the SI-Level, on whose basis the respective measures have been selected, was assumed to be SIL 1. Each section is introduced by a little gray box containing the respective part of the norm the explanations refer to.
Although the intentions of the measures overlap at times, and the sequence in which they are mentioned in the standard does not necessarily correlate with the sequence of their execution in practice, it was decided to leave this order[9] unchanged for easy comparison with the standard. The downside of this approach comes with some redundancy and quite a few cross-references which could have been avoided if the structure of the norm had been neglected.
If only a quick overview over the actual capabilities of the software is desired, the reader is advised to start with section 3.1.3 and concentrate on the external HTML-documentation thereafter.

Additional guidelines or papers for documentation purposes, such as those required for *Modelling* [CEN11, A. 17, p. 76] or for the *Coding Style* (section 3.2.4), have not been produced.

3.1 Software architecture

Table:
A. 3, page 69
➜ EN 50128

This set of measures forms a part of the *Architecture and Design* requirements [CEN11, sec. 7.3, p. 40] focused on "minimizing the size and complexity of the safety part of the application" [CEN11, l. 1373], and understanding the implications of these remaining parts by analyzing them on a fairly abstract level. The techniques utilized should be sufficient to identify all components of the software [CEN11, l. 1376] and thereby form a useful input document to the subsequent set of less abstract measures discussed in section 3.2.

[8] However, the phases are not limited to these tables. On that score the titles of sections 3.1 and 3.2 differ slightly to correspond exactly with the the naming of the tables.
[9] The upcoming sections are ordered according to the *approved combinations* in the respective tables of sections 3.1 and 3.2 with the mandatory measures first and the optionally required ones second.

3.1.1 Defensive Programming

Fulltext:
D. 14, page 97
➜ EN 50128

This requirement is divided into two parts. The first half entitled *intrinsic error-safe software* deals with measures to make software safer "under the hood". Fortunately, all points mentioned here, such as *range-* and *dimension-checking*, are standard functionality of the Ada language. To illustrate their use see the following code example:

Listing 2: Explicit conversion from bytes to numbers in Ada

```
1  function ToNatural (Input: in STREAMLIB.Stream_Element) return Natural is
2     subtype Source      is STREAMLIB.Stream_Element;
3     type Naturaloutput is new Natural range 0..255;
4     for Naturaloutput'Size use STREAMLIB.Stream_Element'Size;
5     subtype Target      is Naturaloutput;
6     function convert    is new Ada.Unchecked_Conversion(Source, Target);
7     Result: Natural;
8  begin
9     Result := Natural(convert(Input));
10    return Result;
11 exception
12    when others =>
13       raise CONVERSION_ERROR with "Cannot␣convert␣to␣numerical␣value.";
14 end ToNatural;
```

This little helper-function inside *Adaimageprocessor.Image* is by far the most frequently called function in the entire client-program. Its purpose is to convert byte values received from the server via the network into numbers, which represent grayscale pixel values. So it has to be called for each and every pixel of the image to be analyzed. As lines 6 suggests, the code is based around an unchecked conversion which actually circumvents Ada's strict type checking, and therefore is dangerous in its nature. But in this case it is truly necessary to convert apples to oranges, since a byte value is simply not the same as a number. Ada allows for this kind of type casting, it just asks for an extremely verbose declaration of it. Coping with the side-effects possibly implied by such an operation is essentially what the standard requires the programmer to do.

To start at the very top of the example code, the standard dictates to check for *access permissions* to parameters. This is done in line 1 via the **in** identifier which explicitly makes a parameter read-only, and the **return** keyword which implicitly grants write-only access. *Type, dimension and range-checking* for these two values are also implicitly handled by the Ada-runtime.
For the variable type Naturaloutput, whose only purpose is to force Ada into runtime range-checking, the upper and lower boundaries of a valid pixel value (line 3) as well as the physical size of such a value in memory (line 4) are explicitly stated[10]. Hence, this satisfies not only the requirement toward *range-checking* but also toward *plausibility*. In line 9 the result is cast into a normal Natural, which is a base type of Ada with much looser boundaries. This has proven more convenient for the further processing of the pixels than a dedicated variable type.
If any errors in between lines 2 and 10 are detected, they are directly handled by the exception in lines 11-13 which will print a descriptive error message and cause the program to fall into a

[10] For a standard computer when 8 bits equal 1 byte this statement is redundant, but for more unconventional systems in which this is not the case it would rightfully introduce an exception. See also `Setup()` in *Adaimageprocessor.Network.Protocol.Imagetransfer*.

safe state, which currently means shutting down all running parts.
For the general use of *Ada.Unchecked_Conversion* see also [Sof95, sec. 5.9.1].

The second part of the requirement focuses on the communication with the outside world. Since the software presented here does not directly interface with any hardware parts, only the interaction with the operating system can hold for an appropriate example. The most error-prone part here is the acquisition of resources. Consider the following snippet from the server program (found in *socket.c*):

Listing 3: Binding to a socket in C

```
1  if (bind(sockethandler, (struct sockaddr *) &ownaddress, sizeof(ownaddress
       )) < 0)
2       {
3         error(__FUNCTION__, "Bind error.");
4       }
```

This very simple if-construct will cause the server to bind (connect) to a given socket only with exclusive access, thus preventing any forks of itself or other programs which would try for the same. This is crucial because otherwise the operating system would not know to whom the data received on the socket should be forwarded.
The last demand by the standard is to implement an inherent check by the software for self-completeness. Currently, there is no such functionality in the client-program present. However, a basic check for the availability of all modules including not only their headers but also their actual code is automatically undertaken by the Ada compiler.

3.1.2 Fully Defined Interface

Fulltext:
D. 38, page 111
➜ EN 50128

Despite its different name, the requirement refers to the exact same fulltext as the "Modular Approach" covered in section 3.2.2 below. In practice, different teams would be responsible for the different stages of the V-Model (Figure 6) which this requirement is associated with [CEN11, sec. 5.1.2.11]. This is why it makes sense to list it twice and have it implemented redundantly by different people at different stages of the project.

3.1.3 Structured Methodology

Fulltext:
D. 52, page 117
➜ EN 50128

As the client-program is currently composed of roughly 22600 SLOCs grouped into 50 different functions[11], with the server adding another 750 SLOCs and contributing over 20 functions to the set, the chances of losing track of their interoperation and of the data flow from one to another are fairly high, especially when projects are becoming even larger.

[11] For this section the term *function* is universally used for any coherent part in a program. For C this basically translates directly into the programmatic structure of a *function* while in Ada it may also refer to *procedures* and *(protected) tasks*.

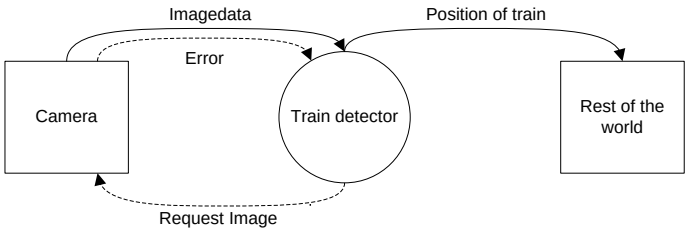


Figure 7: A context diagram as required by the Yourdon-method

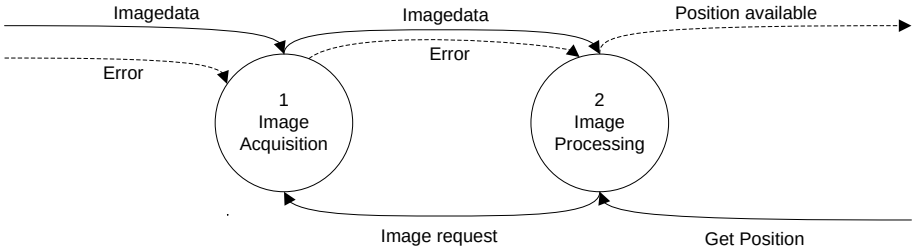


Figure 8: A top level data flow diagram as required by the Yourdon-method

This requirement, which is not only applicable to software development but also to other highly complex processes and machinery, asks for a structured approach in order to illustrate dependencies and hierarchies between functional blocks so that chaos can be avoided right from the beginning. Implementations exist in the form of different methodologies:
MASCOT was developed by the British Army and during its active use in the late 1980s and early 1990s it suffered from little adoption in civilian projects. *LSDM*[12] and especially its non-commercial counterpart *SSADM* have remained in common use until today. However, they are not very well suited for concurrent real-time systems. Hence, only *JSD* and the *YOURDON*-method remain a worthwhile choice in EN 50128 [CEN11, D. 52]. The latter will be discussed in detail further down in this section.
All of these methodologies originally date back to the 1970s when planning and structuring was substantial to any kind of software development, since reading in from punch cards for a second time after detecting a mistake in the code was extremely cumbersome. Today, the methodologies have been largely superseded by the ubiquitous UML which will be utilized in a different context in section 3.1.5 below. Moreover, current *agile approaches* to programming based on frequent software releases and only small incremental steps in between, such as *Scrum* or *XP*, heavily contradict the *waterfall model* (see section 3), which all of the methodologies mentioned above are based on [Sch09, p. 91].

The central idea in *Yourdon*-Modelling used here for the exemplification of the requirement is the *functional decomposition* of the individual software parts in a strictly top-down manner in order to eventually provide "a complete, unambiguous specification of what a system is to do" [Pos86, p. 223]. In the first step a very abstract view is created, subsuming the entire system in a single process ("bubble") and indicating its interfaces to the outer world. This is called a *context diagram* (Figure 7). In the next step this single process is divided into several separate ones describing each part of the system with arrows symbolizing their interactions, a so-called *top-level data flow diagram* (Figure 8). Similar to the *context diagram* continuous arrows denote data

[12] Note: EN 50128 erroneously refers to "LBMS" here, which is just the name of the company that invented LSDM.

flows, whereas their dashed versions stand for control flows which only transmit simple tokens such as Boolean values. Depending on the complexity of the system, this *top-level data flow diagram* may now be broken down further into a *second-* or even *third-level* diagram until finally each "bubble" becomes a *primitive process*, i.e. something so simple it cannot be decomposed any further. Figure 9 shows this for the server (which was named *Image Acquisition* in Figure 8 because implementation details are not yet revealed at this level of abstraction). The numbering attached to each process is not mandatory for the diagramming but it does simplify the correct assignment of the "bubbles" in the context of the overall system. With *Image* a new kind of entity symbolizing a data storage has been introduced here.

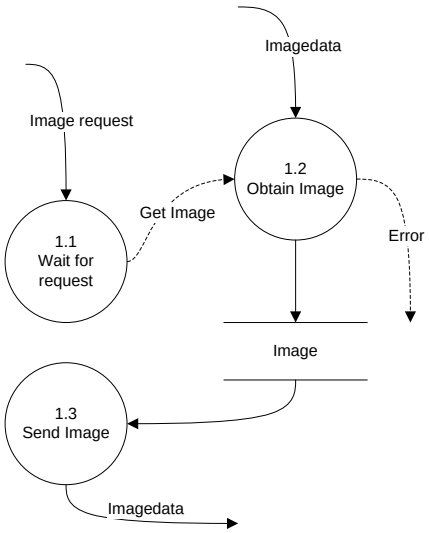


Figure 9: A second level data flow diagram depicting the Image Acquisition process of figure 8

Based on this final diagram, a *data dictionary* is developed which explains the internal structure of each data flow and data storage (Listing 4). Lines 1–6 list them in form of an assignment, with the primitive parts of the respective entity shown on the right. These primitive parts are then further defined in lines 7–12. Several different forms of notation exist, for this example a concise version has been chosen [WM85, p. 100f.] which is similar to that of regexes: Curly brackets (lines 1, 2 and 8) denote the repeated use of a primitive, the plus sign (lines 3 and 7) stands for a concatenation, asterisks (lines 4–6, 9, 11) mark comments including those referring to the unit of data of the current entity. Comments may also be empty if no valuable information can be added (line 4). Square brackets (lines 10 and 12) symbolize an aggregate, i.e. in line 10 a number may consist of a value in between 0 and 9. For line 12 the pipe symbol allows for several such ranges with mutual exclusion, i.e. a character may either be a number between 0 and 9 or a colon or in fact a letter of the alphabet.

Data dictionaries visualize the structure of data flows in a very sententious way. However, complex structures are hard or even impossible to show (there is a maximum length of characters to the `Error`, the `Pixel_Tupel` consists of exacly one X- and one Y-`Pixel`, units can only be denoted in comments, `Imagedata` and `Image` differ internally, with the former only consisting of parts ["chunks", section 2.2] of the latter etc.). Hence, *minispecs*, short verbal descriptions of what a process does with its data, have been invented to overcome this drawback. They consist of identifiers from the *data dictionary*, imperative verbs ("do", "get", . . .) and reserved keywords ("if", "otherwise", . . .) to make up simple sentences which state decisions and/or repetitions [PJ95, p. 195f.].

Consider the following example spec from process 1.3 depicted in Figure 9:

1.3 SEND IMAGE
Initialize an offset with 0.
For each call, do the following:
Write the current callcount to the first 4 bytes of "Imagedata".
Initialize an offset2 with 5.

For the bytes of "Imagedata", do the following:
Take the data from "Image" at offset and write it to "Imagedata" at offset2.
Increment offset and offset2.
If there is no data left in "Image" or "Imagedata" is full, stop the iteration.
Otherwise: Repeat.
Return "Imagedata".

There is no definite syntax for the so-called *structured English* which was used in the *minispec* other than the guidelines mentioned above. Moreover, those specs require strict length boundaries for the individual processes or they will soon become confusing due to their lack of grouping. For instance, in the above text, which maps to the function `protocol_TransmitChunks()` in *protocol.c*, it is not formally stated where the two loops declared by the `For`-keyword effectively end. The concrete spec should be unambiguous, nevertheless. But this claim cannot be generalized.

Listing 4: A data dictionary for Yourdon-Modelling based on Figure 9

```
1  Imagedata = {Pixel_Data}
2  Error = {character}
3  Image request = Pixel_Tuple + Pixel_Tuple
4  Get Image = **
5  Image = *current image*
6          *units : aggregate of 8 bits*
7  Pixel_Tuple = Pixel + Pixel
8  Pixel = {number}
9          *units : positive number in the range of the imagewidth/-height*
10 number = [0-9]
11 Pixel_Data = *units : aggregate of 8 bits*
12 character = [A-Z|a-z|number|:]
```

Yourdon's last step deals with the actual implementation of the software. For this purpose *Structure Charts* are to be drawn reflecting the hierarchical arrangement of the individual functions and their interactions. As they give very concise insight about how a program is constructed, these charts have not only been created on the basis of one single example but for all relevant parts of the software. They are depicted in Figures 10, 11, 12, 13 and 14 on the following pages. Rectangles symbolize individual functions. Their names ought to be consistent with the *data flow diagrams* discussed above. However, since those were only drawn on an exemplary basis, names of the actual functions as they appear in the software have been preferred here. Those rectangles are connected by arrows which denote a subfunction call. In case of dashed arrow lines (as in Figure 10) this call is invoked asynchronously. Despite the inherent orientation of the arrow, control flow may also happen in reverse direction once those subfunctions return. Little pointers adjacent to those arrows stand for data which are exchanged between the functions. They may come with a circle- (arbitrarily complex data structures) or a bullet-tail (simple tokens as discussed above). Arrows whose beginning and end are attached to the same function indicate repetitive execution. Triangles on top of the rectangles make them "lexically included" with their parent. Here, this is used for parent-functions which do not exist in the actual software but either contain mutually exclusive decisions denoted by a little diamond at the lower end of the rectangle, or simply facilitate the understanding of the grouping of their subfunctions. In Figure 12 there is also a rectangle with double lines along the vertical axis which stands for a predefined module (i.e. a call to `Put_Line()` in *Ada.Text_IO*). Finally, Figure 11 contains a rectangle

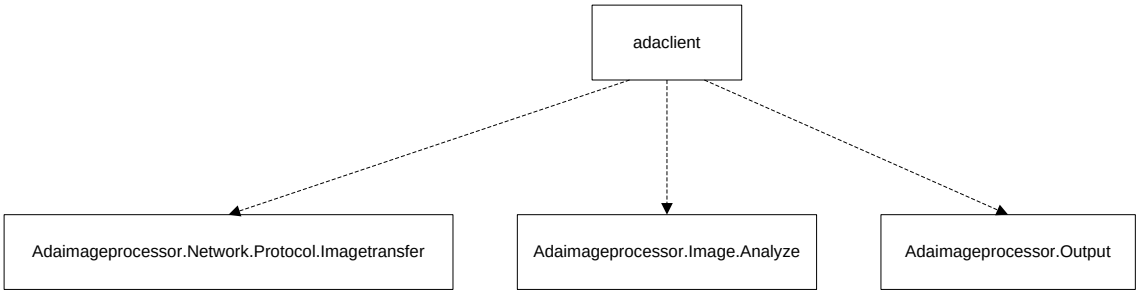


Figure 10: A structure chart of the client-program as required by the Yourdon-method

with vertical lines rounded to indicate a data-only module. For an in-depth discussion of the different symbols refer to [YC79, app. B].
The main drawback of these *Structure Charts* is their lack of sequence. It is not defined in which order the functions are called other than by the implicit presence of required input-values computed by other functions before. With asynchronous calls as in Figure 10 this is unproblematic (in fact all these Ada-tasks represented by the rectangles are started in parallel). This drawback can only be overcome by the use of other diagramming techniques such as the UML *Sequence Diagram* discussed in section 3.1.5.
Furthermore, the definition of a *function* or *module* which the rectangle is supposed to stand for, does not necessarily match with the capabilities of a certain programming language. For the outermost level in the client-program the name of the respective Ada packages has been used. However, for Figure 10 this would imply a call of such a package which – strictly speaking – is incorrect. The packages are rather **with**ed which implicitly causes their individual tasks to start. If one package contains only one task, this notation is unambiguous. But for *Adaimageprocessor.Image.Analyze*, which contains several such tasks including the two parallel rail-workers (Figure 15), this may lead to severe confusion.

Edward Yourdon, a computer engineer after whom this methodology was named, always conceived it not only as a set of tools and diagrams but, indeed, as a philosophy to follow throughout the entire development lifecycle. For him, it was a structured way of thought to approach problems. Hence, for him there is much more to the whole story than what can be shown on a few pages targeted only at reaching a compliance level with EN 50128. See [YC79, sec. 1.2].

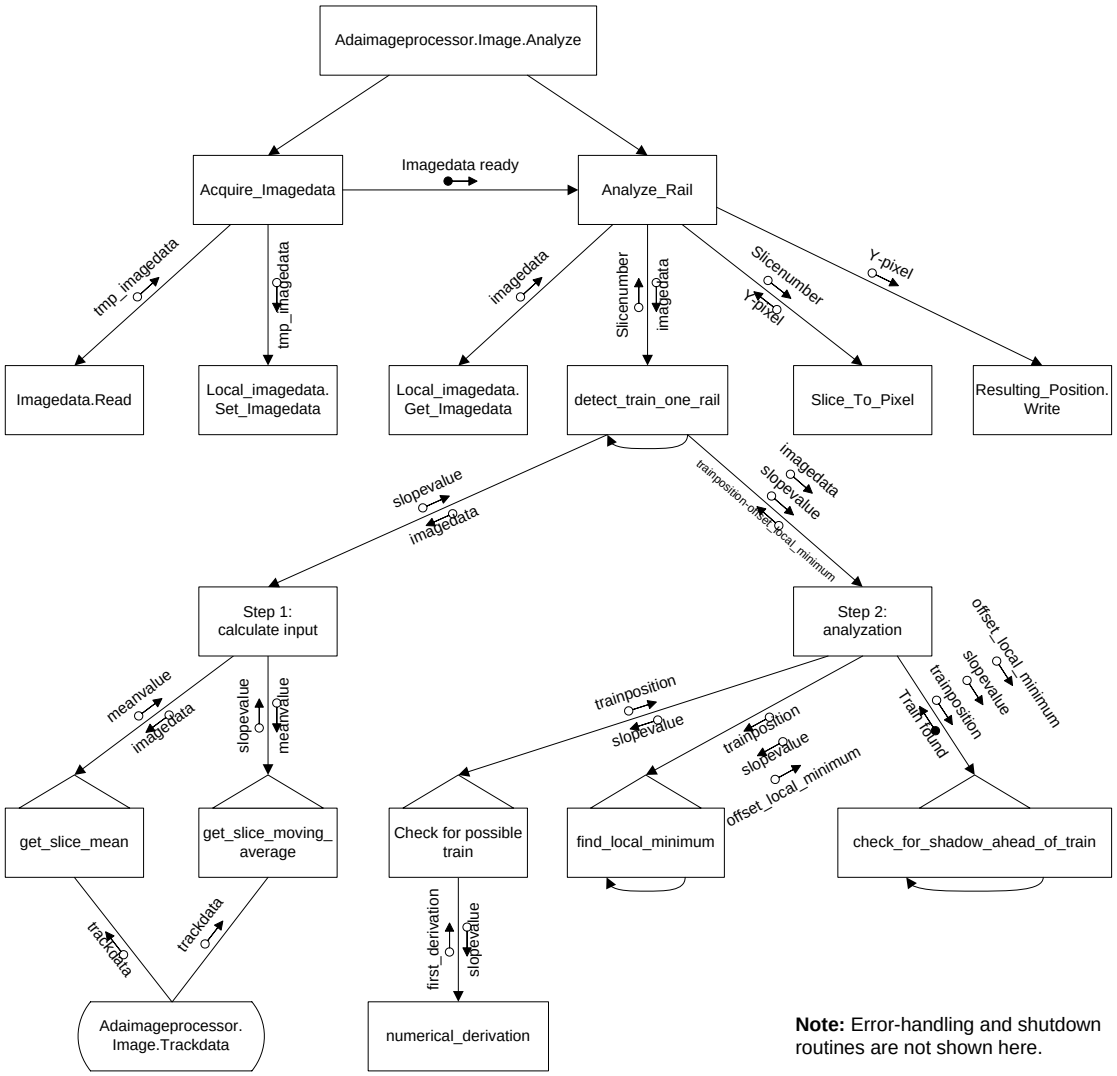


Figure 11: A structure chart of the imageanalyze task in the client-program as required by the Yourdon-method

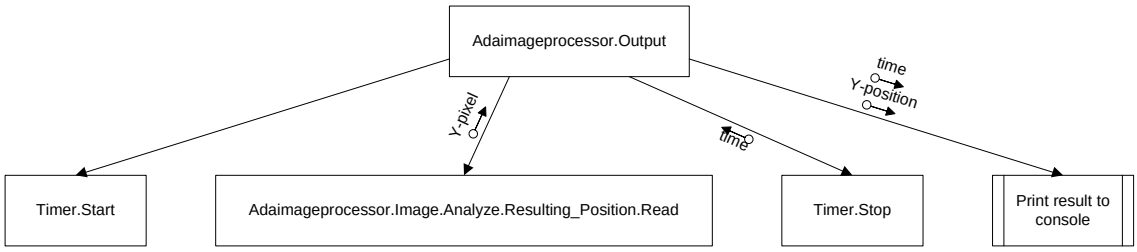
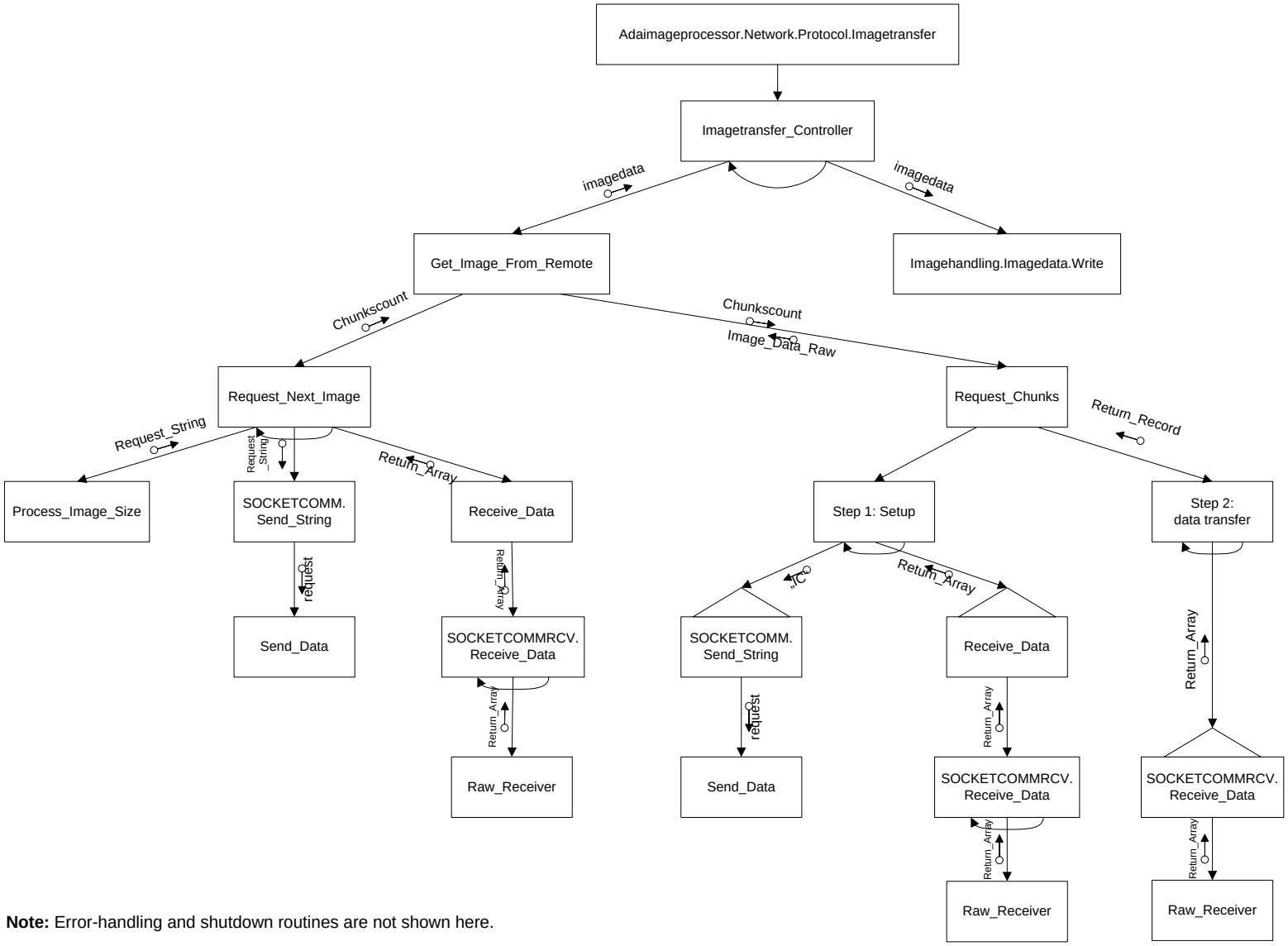
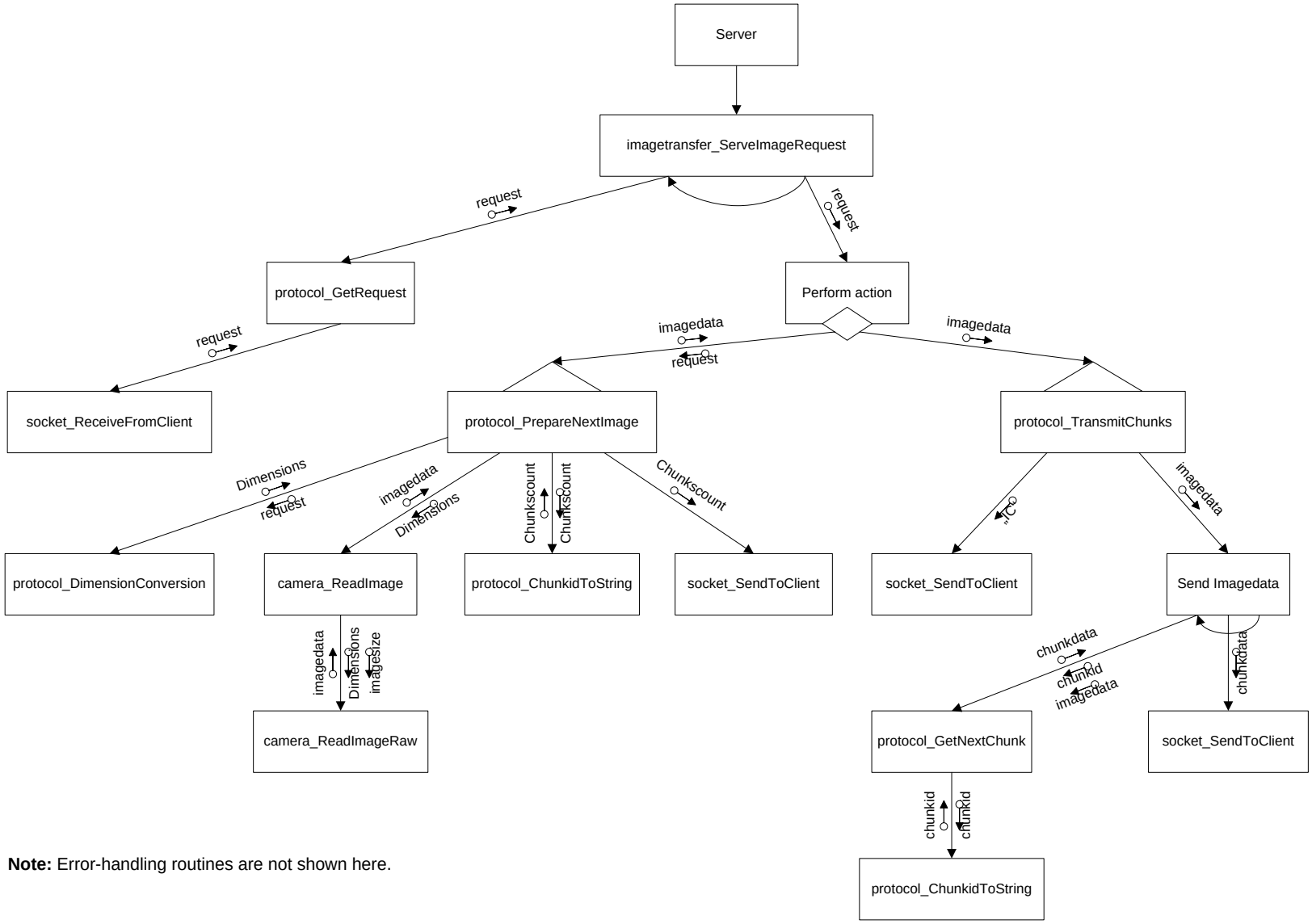


Figure 12: A structure chart of the output task in the client-program as required by the Yourdon-method



Note: Error-handling and shutdown routines are not shown here.

Figure 13: A structure chart of the imagetransfer task in the client-program as required by the Yourdon-method



Note: Error-handling routines are not shown here.

Figure 14: A structure chart of the server-program as required by the Yourdon-method

3.1.4 Error Detecting and Correcting Codes

Fulltext:
D. 19, page 98
➜ EN 50128

This requirement is supposed to prevent any unintentional alteration of data. Since such alteration may occur even at hardware level with its effects propagating upstream until they eventually lead to anomalies in the safety-critical software, it is suggested to use functions which store a footprint of the data and can later compare this footprint to the current state of the data whenever they are read, thus ensuring their integrity.
Easily computable, some of these functions may be directly implemented in hardware and mainly provide protection against physical influences (e.g. electromagnetic interference). RAM with "error-checking and correcting support", the so-called ECC-RAM, and the UDP-checksum discussed below, fall into this category. Other more complex algorithms, especially those of cryptographic origins, primarily recommend themselves for an implementation in software and may also protect from man-made attacks intended to corrupt the system.
A famous example of successfully exploiting software-based footprinting is Stuxnet, a computer worm revealed in 2010 which was targeted at Iranian nuclear power plants running real-time, safety-critical control code on Siemens Simatic S7 PLCs [Lan10]. In order to infiltrate these automation systems the worm first had to gain full execution rights on the computers used to program them. To accomplish this it turned itself into a scheduled task[13] and altered the control file associated with this task in such a way that it gained the necessary rights on the computer it was running on. Preventing the operating system (Windows Vista and higher) from detecting this manipulation, the worm also changed the checksum for this file, which was in the form of a CRC32 hash, to match the original sum. This can be done by appending certain bytes to the file until its polynomial representation (which is used as the CRC input dividend) becomes a multiple of the CRC generator polynomial (a constant value used as the divisor), constituting an operation which forces a collision with the original quotient (the actual CRC hash) and hence a (false-)positive data integrity check [SPMR06, sec. 3]. Microsoft, the vendor of Windows, later replaced CRC32 with SHA256 for those control files to stop future attacks of this manner [Dan10, time 17:38]. By the definition of EN 50128, CRC checksums fall into the category of *cyclic codes*, whereas SHA belongs to the family of *cryptographic codes* [CEN11, D. 19].
According to a recent statement by former NSA employee Edward Snowden, Stuxnet is believed to have been heavily backed by the American and Israeli governments in order to purposely delay Iranian Nuclear Weapon proliferation [AP13]. Its success remains controversial.
What the above example impressively shows is that hash algorithms with a high probability of collisions (i.e. different input values result in the same output hash) are not very well suited for detecting data corruption and their use should therefore be carefully justified.

A good example of the incorporation of the requirement in the software developed for this project, the UDP-based network transmission (see section 2.2) is targeted at preventing physical data corruption. Each single network packet comes with an 8 bit wide checksum generated in hardware on the NIC of the sender and checked thereafter by the NIC on the receiving side. If the check fails the packet is silently discarded. This makes the checksum only an *error detecting* but not an *error correcting* code.

[13] Not to be confused with Ada tasks. Here *task* has a more general meaning and essentially refers to everything that can happen periodically.

	part	meaning	value	value as 16 bit Hex
1	pseudo header	Source IP	192.168.1.200	`0xC0A8 0x01C8`
2		Destination IP	192.168.1.15	`0xC0A8 0x010F`
3		UDP protocol type	17	`0x0011`
4		Packet length *[bytes]*	11	`0x000B`
5	UDP packet	Source port	12345	`0x3039`
6		Destination port	33990	`0x84C6`
7		Packet length *[bytes]*	11	`0x000B`
8		Checksum as zeros	0	`0x0000`
9		payload data	IC\0	`0x4943`

Table 1: Relevant fields for the UDP checksum calculation

To illustrate how the checksum computation works consider packet number 4 from Table 2, respectively Figure 16. The physical structure of this packet just before it is transported over the wire is depicted in lines 5 through 9 of Table 1. Lines 1 through 4 form a so-called *pseudo header* of UDP. In contrast to the physical header of lines 5 through 8 it is not actually transmitted but rather composed of fields belonging either to the physical UDP-header itself or the IP transport layer (see section 2.2). Its only purpose is the checksum computation.
To actually execute the computation first convert every byte to its hexadecimal representation. Then group every two consecutive bytes together. The result is shown in the last column of Table 1. Now take the sum of all these values (`0x28290`) and add the higher eight bits (`0x0002`) to the lower ones (`0x8290`). The result (`0x8292`) must finally be complemented (`0x7D6D`) to form the checksum. Substitute it for the dummy value in line 8 of Table 1 and send the packet.
The receiver again has to sum up all values in the last column of Table 1, but now with line 8 being the real checksum and no longer a zero value (`0x2FFFD`). After that, the high and low halves are added (`0xFFFF`). This result is defined to be correct by the underlying standard [BPB88]. All other outcomes would cause the packet to be discarded.

As stated before, this checksum is only intended to prevent physical damage to the data. For any purposely altered payload (Table 1, line 9) simply a new and correct checksum could be generated. Referring to the case of Stuxnet, this does not protect in any way from people intentionally intercepting and corrupting the transferred image pixels (a so-called "Man in the middle"-attack).

3.1.5 Modelling

Table:
A. 17, page 76
➜ EN 50128

The requirement toward modelling lists eleven different techniques, two of which are marked as *highly recommended* for SIL 1: *State Transition Diagrams* and *Sequence Diagrams*. Even though only one of those two would be sufficient for the fulfillment of this requirement, both can be illustrated very well by the software created for this project and hence will be discussed here. Some of the remaining *recommended* modelling techniques, such as *Structure Diagrams* and *Data Flow Diagrams*, have been shown elsewhere in this document when talking about the Yourdon-method in section 3.1.3.

A *State Transition Diagram* may be modelled in a number of different ways. For reasons of conformity with the *Sequence Diagrams* discussed below, an UML *State-Machine Diagram* has been chosen here. It supplements the structure charts used for the Yourdon-method (section 3.1.3) by giving information about the order in which certain blocks of code are to be executed. The basic idea behind such diagrams lies in the different states a program may be in. With the exception of *orthogonal regions* (see below), only one state at a time may be active.
The example in Figure 15 depicts a view of the task-level[14] on the client-program. The circles describe *entry*- and *exit*-points to and from the system. The arrows in-between them show the control flow. The text in square brackets that may come along with those arrows is called a *guard* and contains a Boolean condition which has to evaluate to *true* before the flow continues. In the case of Ada these guards translate either to *semaphores* (*[Analyze_Semaphore. . .]*) or to *guards* of protected tasks (all other cases). Both of these structures are very limited in their functionality due to the use of the Ravenscar-profile (refer to section 3.2.6). *Client program* and *Train detection* form so-called *superblocks* which group certain cases of functionality. For the latter, this superblock contains two *orthogonal regions* to depict the concurrent nature of the train detection which involves both rails being examined in parallel. However, the comparison of the outcome of both detection algorithms is hidden inside the *Resulting_Position* object and thus not shown in this very abstract diagram.
All states (the rectangles with rounded corners) may come with *entry*-, *exit*-, and *do*-actions which describe what has to be done when entering, leaving or while remaining in this state. The snapped rectangles are used for commentary.

Another member of the family of UML dynamic diagrams, the *Sequence Diagram*, can be seen in Figure 16 exemplifying the UDP-based protocol used for the transmission of data between the server- and the client-program (see section 2.2). The central aspect is the communication between different entities. In the case of Figure 16 these are simply *:Client* and *:Server*. The *lifelines* attached to them (dashed vertical lines spanning over the entire diagram) depict the temporal course in which actions are executed by each entity. Phases of *activation* (longish rectangles on the *lifeline*) indicate specifically when this happens. Due to the connectionless nature of UDP all the requests by the client (arrows pointing to the right) are asynchronous and hence end with only half an arrowhead. Server responses, in contrast, are symbolized by left bound dashed arrows. Each of these arrows represents a specific packet of the underlying protocol, whose internal structure is shown in Table 2. Blocks entitled with *loop* or *alt* stand for either recurring operations or decision alternatives. Both come with *guards* in the same notation as for the *State-Machine Diagram* discussed above. For the former these guards describe the looping-condition (continue as long as the guard evaluates to *true*), for the latter they decide upon which alternative is chosen (the upper if the guard is *true*, the lower if not).

In summary, the *Sequence Diagram* in Figure 16 is located at a much lower level of abstraction than the *State-Machine Diagram* in Figure 15. Specifically, the former describes only a part of `Get_Image_From_Remote()` which is the entry point in the *Imagetransfer_Controller*-state of the latter.

[14] In the sense of an Ada **task**.

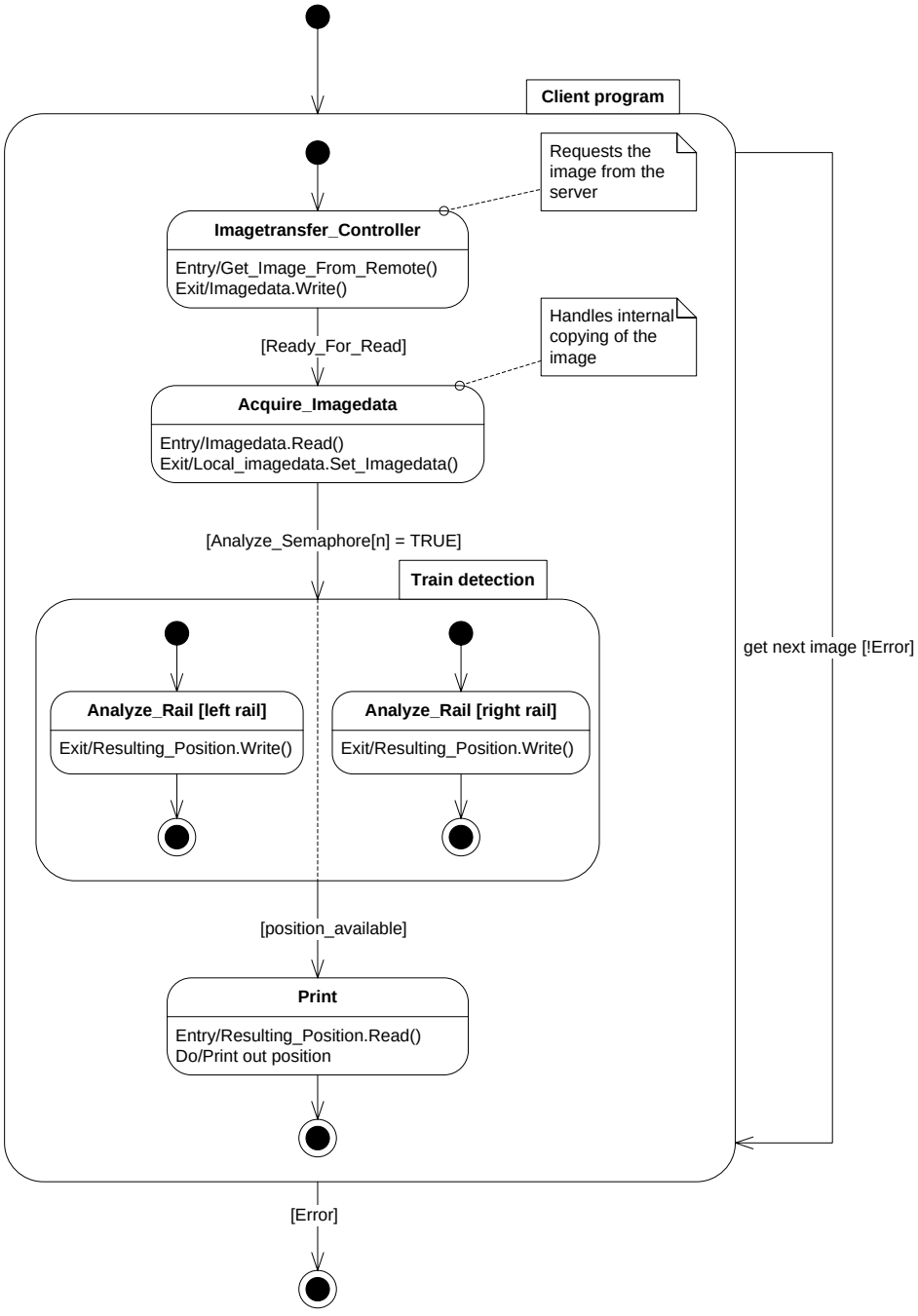


Figure 15: An UML statechart diagram of the client-program

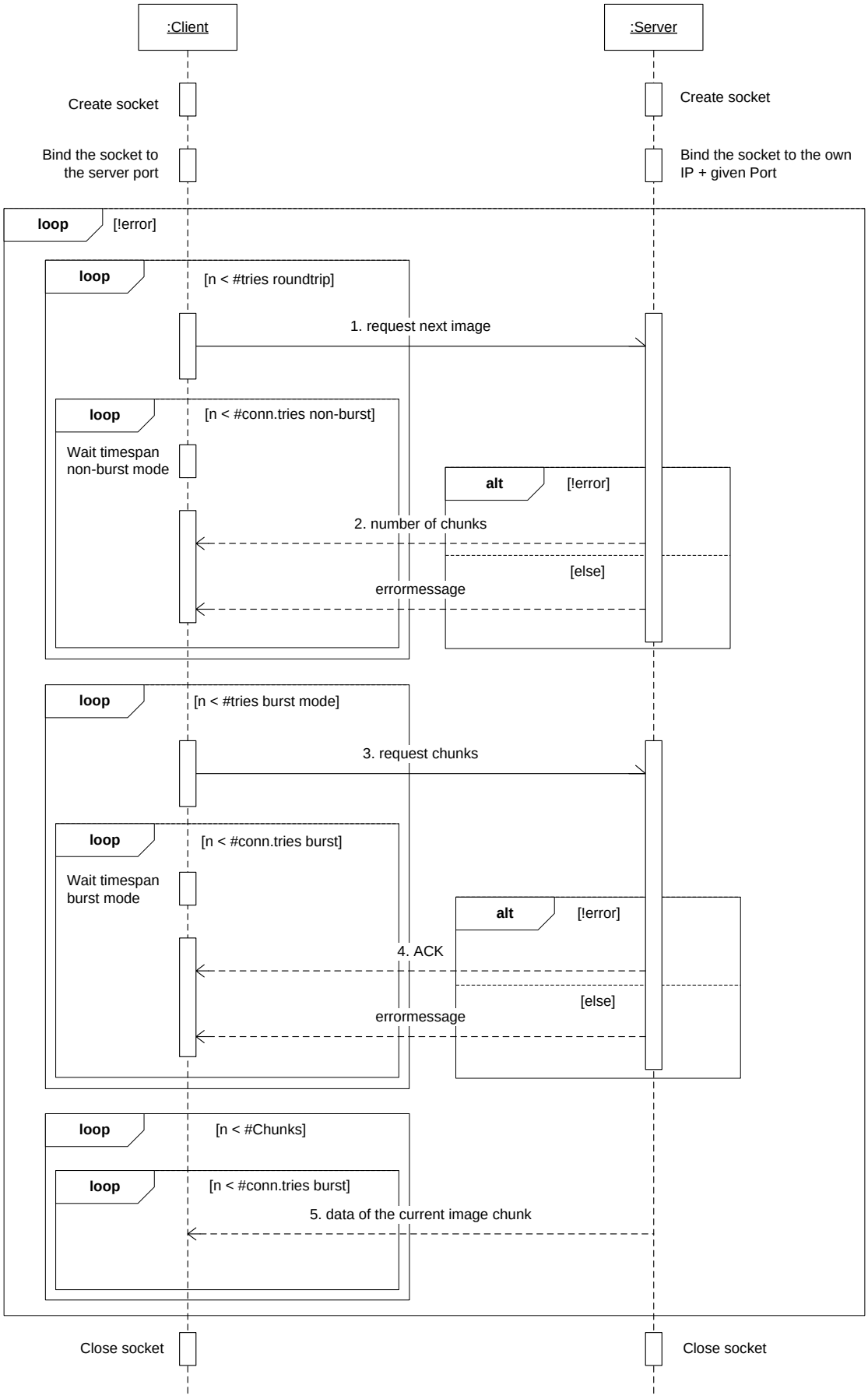


Figure 16: UML Sequence Diagram of the UDP-based protocol

	name of step	structure of the respective packet
1	request next image	`IN` *<top left X padded to 4 characters> <top left Y padded to 4 characters> <bottom right X padded to 4 characters> <bottom right Y padded to 4 characters>*
2	number of chunks	`IN` *<number of chunks padded to 4 characters>*
3	request chunks	`IC`
4	ACK	`IC`
5	data of the current image chunk	`<sequence number padded to 4 characters>` *<data>*
	errormessage	`ER` *<errormessage trimmed to 53 characters>*

Table 2: The structure of the packets associated with the different steps shown in Figure 16

3.1.6 Alternative optionally required measures

Annotation number 2 of the standard's table referred to in section 3.1 lists seven optionally required measures one of which has to be selected. As *Error Detecting and Correcting Codes* (section 3.1.4) and *Modelling* (section 3.1.5) are among them, the coverage in the sections above was, strictly speaking, more than sufficient.
Nevertheless, here is a short summary of the other possible measures including the reasons why they have not been taken into account for this project.

Table:
D. 26, page 101
➜ EN 50128

Fault Detection and Diagnosis requires a system to deliver either correct results or no results at all. This must be achieved by *assertion programming*, *N-version programming* or the *Safety-Bag technique*. The first two are mentioned en passant further down in section 3.3 with the *assertion* corresponding to the Assert statement provided by `aunit` and the *N-version* to the testing code presented in Listing 9 which computes the same result as the code to be tested differently. However, a general use of *N-version programming* outside of the testing-routines would require additional developers and/or a different programming language to proof useful. Neither was feasible for this project. Likewise, the *Safety-Bag technique* which asks for an external supervisor (i.e. a second computer) [CEN11, D. 47, p.115] would have required too much effort here.

Fulltext:
D. 24, page 100
➜ EN 50128

Failure Assertion Programming is headed in the same direction as the *assertion programming* mentioned in the previous paragraph. To extend a little upon it, SPARK [Alt11, sec. 3.4] and the most recent incarnation of the Ada language (Ada 2012) provide special pre- and postconditions to functions and procedures which allow for an easy way of assertion programming (see section 4). However, SPARK is aimed at much higher SI-Levels, and Ada 2012 features have simply not yet been incorporated into the client-program.

Fulltext:
D.,16, page 97
➜ EN 50128

Diverse Programming is merely a combination of the *N-version programming* and the *Safety-Bag technique*, both mentioned above. Unfortunately, benefits of applying this extensive requirement in the field have remained doubtful until this day. In summary, there is no guarantee that a certain algorithm being implemented by different people on different computers, running different operating systems will provide any more reliable results than a single one on a single machine. Instead, the overhead resulting from the comparison of their results might even cause new faults.

Fulltext:
D. 36, page 110
➜ EN 50128

Memorising Executed Cases requests for a set of predefined execution paths against which the running program is compared before any output operation takes place. As there are no real large-scale decisions in the client-program (confer with the structure charts of section 3.1.3), compared to an interlocking-system for instance, and the programmatic check against the validity of those execution paths is not trivial, the measure has not been considered.

Fulltext:
D. 25, page 100
➜ EN 50128

Software Error Effect Analysis is an application of the FMEA on software development. As software may only contain systematic faults [Red06, p. 339], the analysis essentially boils down to three stages: Determination of the necessary thoroughness of the examination (consider each line of code or just each function?), identification of the errors possibly resulting from those entities and eventually the weighing of those errors, combined with a presentation of hazardous scenarios ignored by the current software implementation. As this is required to be carried out by an independent team of examiners and demands for a lot of paperwork, it has not been considered for this project, either.

3.2 Software Design and Implementation

Table:
A. 4, page 70
➜ EN 50128

Although this set of measures is already referenced in the section about *Architecture and Design* of the standard [CEN11, sec. 7.3.4.24, p. 44], it does officially belong to the subsequent section titled *Component Design* [CEN11, sec. 7.4.4.6, p. 47] (confer with the V-Model in Figure 6). The measures are generally more specific to the actual coding of the software than those discussed previously (section 3.1). Their aim is to identify whether the coding will be feasible and the result maintainable and testable [CEN11, sec. 7.4.4.13, p. 48], with the last requirement being a prerequisite for the following section 3.3.

3.2.1 Structured Methodology

Fulltext:
D 52, page 117
➜ EN 50128

This requirement exactly doubles with the one from section 3.1.3. However, the statement from section 3.1.2 regarding intentional redundancy does not apply here, since the *Structured Methodology* is more of a fundamental approach to software development which simply has to be followed throughout the entire development process as outlined by the V-Model in Figure 6.

3.2.2 Modular Approach

Fulltext:
D .38, page 111
➜ EN 50128

Contrary to most other requirements of EN 50128, which state their goals in a rather abstract, implementation-independent form, this particular one names eight specific points to be taken care of. However, due to the comprehensiveness of some of them, additional measures such as those outlined in section 3.2.4 should also be carried out.
The aforementioned eight points are printed here verbatim in bold letters, with their explanations following.

a module shall have a single well defined task or function to fulfil This has already been stated as a requirement of the so-called *minispecs* in the Yourdon method discussed in section 3.1.3 and can therefore be seen as inherently fulfilled.

connections between modules shall be limited and strictly defined, coherence [...] shall be strong This is also required by Yourdon (section 3.1.3) and implicitly enforced by the Ada programming language and its fine-grained variable scopes as well as its Ravenscar-Profile which requires the programmer to use *protected* objects to avoid race-conditions with different tasks and thus implicitly limiting connections between them.

collections of subprograms shall be built providing several levels of modules Ada enforces this through its sophisticated inheritance features. See the structure charts 13, 11 and 12 in section 3.1.3 for how it is done in the client-program.

subprograms shall have a single entry and a single exit only For the *entry point* and for simple *procedures* and *functions* this is guaranteed by the Ada language itself. Each call inevitably has to start at the very beginning of those subprograms. For *tasks* Ravenscar [Ada06, D.13.1] limits their entry points to none, which means the task can only behave like a procedure. *Protected* objects may have one single entry with a maximum of one waiter. Both limitations lead to full compliance with the first part of the requirement (see also section 3.2.6).
The demand for a single *exit point* is mostly honored throughout the program as well , but *exceptions* form a notable and deliberate deviation from the rule. The need to quickly react to a malfunctioning software and the attempt to recover from this state should supersede this guideline here. Sometimes several exit points also make an algorithm clearer. Consider the following abridged version of the function `detect_train_one_rail()`:

Listing 5: Shortened version of the train detection algorithm in Ada

```
1  function detect_train_one_rail (...) return Integer is
2     begin
3        for Slice in rail_of_interest'Range loop
4           slopevalue(Slice) := [...];
5
6           get_absolute_minimum;
7           compute_gradient;
8
9           if gradient >= threshold.derivation_upper then
10             possible_train_found :
```

```
11              begin
12                 find_local_minimum;
13
14                 if delta_of_absolute_and_local_minimum_is_small_enough
                      then
15                    check_for_shadow_ahead_of_train :
16                    begin
17                       while slopevaluechange_in_acceptable_range loop
18                          offset_shadow := offset_shadow +1;
19                          if offset_shadow =
                               minimum_sections_for_shadow_occurrence
                               then
20                             return Slice-offset_local_minimum;
21                          end if;
22                       end loop;
23                    end check_for_shadow_ahead_of_train;
24                 end if;
25              end possible_train_found;
26           end if;
27        end loop;
28
29        -- no train detected
30        return -1;
31  end detect_train_one_rail;
```

The function may exit in both lines 20 and 30. This is directly linked to the fact that this function will stop executing as soon as it finds a train (line 20). Only if there is absolutely no train present, execution will proceed until line 30. Of course, the same algorithm could be implemented by replacing line 20 with an **exit when** ... construct[15] and saving the return value in some temporary variable, thus allowing for a single exit at the very end. But in this special case it would not necessarily aid program comprehension[16].

modules shall communicate with other modules via their interfaces. Where global or common variables are used they shall be well structured, access shall be controlled and their use shall be justified in each instance Data common to *tasks* are used in read-only mode exclusively to avoid race-conditions. This can be very well achieved by either getter-functions (i.e. `Local_imagedata.Get_Imagedata` in *Adaimageprocessor.Image.Analyze*) in conjunction with semaphores [Ada06, D.10] or *Protected entries* (i.e. `Resulting_Position.Write` in the same package).
For ordinary *procedures* and *functions information encapsulation* via getters and setters applies as well (see section 3.2.3). In addition, the compiler switch `-gnatwk` (Table 4) was used to make sure all constant values are always declared constant.

all module interfaces shall be fully documented Extensive documentation covering parameters, discriminants and return values of each module is provided in HTML–form.

any modules interface shall contain the minimum number of parameters necessary for the modules function Ada guarantees this automatically. Only simpler languages like C with its *extern* statements to declare functions outside of the current file could potentially violate this requirement. But even for the server, which is written in C, this has

[15] Or simply a *goto* – but this is considered the double return's equally evil twin. Refer to section 3.2.4.
[16] The Ada Quality and Style Guide generally recommends the *exit when*-variant – but does allow for the *if/then*-construct as well [Sof95, sec. 5.6.5].

at least been excepted partly by using gcc's `-Wimplicit-function-declaration` and `-Wstrict-prototypes` switches. "Partly" because C still does not check if the number of parameters and their types is consistent everywhere, it now only forces the programmer to explicitly state the parameters wherever the function is externally declared and/or called.

a suitable restriction of parameter number shall be specified, typically 5 The motivation for this requirement remains somewhat unclear, since even when sticking to the rule one could simply pass many more parameters hidden inside records or behind access variables[17] to subprograms. Ada, in addition, allows for named parameters to remove any ambiguity in calling a subprogram [Ada06, 6.4] even with many parameters. Nevertheless, the number of five parameters is not exceeded anywhere in the client-software.

3.2.3 Components

Table:
A. 20, page 77
➜ EN 50128

The standard only mentions two *highly recommended* measures here. The first one titled *Fully Defined Interface* was discussed in section 3.1.2, respectively 3.2.2. Therefore, the only technique left to fulfill this requirement is *Information Encapsulation* [CEN11, D. 33, p. 109].
Its aim is to minimize direct access to globally used data and to provide dedicated methods for their alteration instead (see also *module interfaces* in section 3.2.2), with the advantage of limiting code adaptions in case of changes of the underlying data structure. The simplest forms of such *Encapsulations* are so-called *Getter*- and *Setter*-Methods. In Ada those may be grouped usefully in their own subpackages, whose **private**-parts contain the actual data, so that bypassing the access methods is impossible. Consider the example below taken from the specs of *Adaimageprocessor.Network.Socket*:

Listing 6: Information encapsulation in *Adaimageprocessor.Network.Socket*

```
1  package SettingsManager is
2     procedure Burst_Transfer_On ( Chunknumber : in Number_Of_Chunks );
3     procedure Burst_Transfer_Off;
4     function Get_Roundtrip_Tries return Positive;
5     function Get_Connection_Tries return Positive;
6     private
7        SOCKET_TIMEOUT_MAX : constant Duration := 1.0;
8        SOCKET_TIMEOUT_MIN : constant Duration := 0.7;
9        CONNECTION_TRIES_MAX : constant Positive := 2;
10       CONNECTION_TRIES_MIN : constant Positive := 3;
11       ROUNDTRIP_TRIES : constant Positive := 3;
12       connection_tries : Positive := CONNECTION_TRIES_MAX;
13 end SettingsManager;
```

As the name suggests, this package handles the current settings of the socket-connection. During different phases of the data transfer, different timeouts (the period it takes to wait for a reply by the server) and different retry-counts apply (Figure 16). Here, the *Setter*-methods are the procedures in lines 2 and 3, *Getters* are the functions in lines 3 and 4, *encapsulated data*

[17] respectively structs and pointer constructions in C

follow in lines 7 to 12. The burst transfer is active during the actual transmission of the image data, while the less restrictive non-burst transfer is used for the initialization and handshaking which happens prior to that. High-level routines in *Adaimageprocessor.Network.Protocol* call the two procedures during the respective phases of the transfer and immediately afterwards read out the resulting round trip number (the retry-counts returned by `Get_Roundtrip_Tries()`). `Get_Connection_Tries()`, in contrast, is only used by very low-level socket-functions in *Adaimageprocessor.Network.Socket*. The *Setters* in lines 2 and 3 simply alter `connection_tries` in line 12 and set it to either the value of line 9 (non-burst mode) or 10 (burst mode). As `ROUNDTRIP_TRIES` is not changed during the different phases of the transfer, `Get_Roundtrip_Tries()` can remain without such a variable and only returns a constant value.
The *Encapsulation* of the necessary socket-parameters by introducing this package collects all data at one central point and thus greatly simplifies code maintainability and enhances robustness, just as expected by the standard. Other examples of such *Encapsulations* may be found in the **protected** tasks and in various other conventional child-packages such as `Local_Imagedata` of *Adaimageprocessor.Image.Analyze* or `OperationIdentifiers` of *Adaimageprocessor.Network.Protocol*.

A counterexample of the technique is the package *Adaimageprocessor.Image.Trackdata* whose data are by no means encapsulated. However, since it consists only of constant values and there is just one very dedicated method, namely `get_slice_mean()` in *Adaimagprocessor.Image.Analyze*, which actually accesses it, this can very well be tolerated.

Abstract data types, as suggested by the standard, do exist in Ada as well. But a reimplementation (inheritance) of the same functionality for a different data-layout was not required at any point of the project, so no use has been made of them.

3.2.4 Design and Coding Standards

Table:
A. 12, page 73
➜ EN 50128

Code becomes easier to understand and maintain if a certain *coding style* is followed. So the first part of this requirement asks the developer to establish such a style together with a *coding standard* and obey both henceforward. What this paper should comprise exactly, is defined only vaguely by EN 50128, though [CEN11, D. 15]. *Style* refers to what the code looks like. Naming conventions for variables or a requirement toward strict functional decomposition are examples of what falls under this term. A *coding standard* is more of a language-specific analysis regarding its portability (will a code run on different operating systems, will it be understood by different compilers?), possible shortcomings of the language (C has no array-bounds checking) and how to treat them (the programmer should always make sure memory is allocated for a certain array element before writing to it). Restrictions on how to avoid faults caused by this behavior (code must be able to properly handle an access error when reading from an unallocated array element) eventually make it complete.
The second part of the requirement is slightly more tangible and essentially asks for a formal proof that the rules from section 3.2.2, plus the *coding style* and *coding standard* from above are actually obeyed throughout the entire code. The exact measures to be taken care of are given in [CEN11, A. 12]. Points 3 and 4 of that list are already honored by the Ravenscar pro-

file. The other lines which are marked *highly recommended* are printed together with the appropriate rule switches of `gnatcheck` in Table 3. `Gnatcheck` is a part of the GNAT compiler suite which can be used to testify compliance with certain language restrictions and was originally developed to tackle the coding guidelines of DO-178B (section 1). It comes with an extensive set of predefined rules which can be parametrized, chained together and run against the source-code. If the program does not print any output other than its progress bar, everything is fine. Otherwise, it will give the exact location of where the code violated one of the restrictions referenced in Table 3 (i.e. FileX:LineY:ColumnY: **goto** statement found; not allowed because of `+RGOTO_Statements`).

Table 3 states several embraced rules. Those are actually required by EN 50128, but fail against the code for various reasons: `+RForbidden_Pragmas:GNAT`, which comes from the *coding standard* discussed in the very beginning of this section, would prohibit to use of **pragma** Unreserve_All_Interrupts;. This pragma is specific to the GNAT compiler and forces it to forward all interrupts by the operating system to the program. In the case of the client these are SIGINT, SIGTERM and SIGHUP (defined in *Adaimageprocessor*) which should all cause the client to shutdown immediately [Ada12a]. `+RMisnamed_Identifiers:Default` refers to the naming conventions of program entities in the *style guide* mentioned above. Those conventions would replace the *Default*-parameter printed here [Ada13c, sec. 8.1.6.3]. Since a *style guide* was not written for the purpose of this paper and the restrictions implied by *Default* are fairly rigid, the rule remains unused.

`+RUnconditional_Exits` fails because of its inherent simplicity. Consider the example in Listing 7 below (taken from `Request_Chunks()` in *Adaimageprocessor.Network.Protocol*):

Listing 7: Demonstration of an unconditional exit in Ada

```
1  Initialize_Loop :
2      loop
3          Do_Something;
4          declare
5              Process_Indicator : constant Sometype := Somevalue;
6          begin
7              case Process_Indicator is
8                  when OperationIdentifiers.Request_Chunks =>
9                      exit Initialize_Loop; -- fine, lets proceed
10             end case;
11         end;
12     end loop Initialize_Loop;
```

The rule would object to this because of the missing **when** clause in line 9. However, the exit is hidden inside a **case** construct and therefore entirely conditional. Line 9 could easily be changed into **exit** Initialize_Loop **when** True; to comply with the rule, but that would contradict code simplicity.

The parameters of `+RMetrics_Cyclomatic_Complexity`, `+RMetrics_Essential_Complexity`, `+RMetrics_LSLOC` and `+ROverly_Nested_Control_Structures` have been arbitrarily set, so the code complies with them. The last rule is merely a subset of the first two, which describe much more sophisticated measures to identify code complexity. `+RImproper_Returns` enforces single exit points in functions, something which has already been discussed along with Listing 5 in section 3.2.2. `+RMultiple_Entries_In_Protected_Definitions` is enforced by the Ravenscar subset as well and therefore doubles here (see section 3.2.6).

Unfortunately, `gnatcheck` does not provide sufficient checks for clause 10 in Table 3, although

this requirement may sound oddly simple at first. `+RNon_Visible_Exceptions` only covers a very specific case here, which was probably not even thought of by the authors of EN 50128. However, with `adactl` (full program name: Adacontrol, [Ada13d]) a second heavyweight code checker exists providing several checks that can tackle this requirement, namely:

Listing 8: Example of a rule file for use with Adacontrol

```
1 check directly_accessed_globals;
2 search reduceable_scope(variable, type);
3 search global_references(multiple, task, protected);
```

The above notation is in the form of a *rule file* as required by `adactl`, see [Eur13]. The *check* and *search* identifiers describe different consequences of the tests. The former reports every violation as crucial and forces `adactl` to exit with a failure code, while the latter only informs about the violation. `Gnatcheck` uses a similar input file which is a simple concatenation of the rules as specified in the right column of Table 3 with every violation treated as crucial.
Lines 2 and 3 in Listing 8 currently produce a fair number of violation messages. For line 2 this is due to the fact that all variables and types worth of documentation are declared in the *package specs* (the files ending with **.ads*) because NaturalDocs, the tool used to generate the final HTML documentation [Val11], currently does not process the *package bodies* (the files ending with **.adb*) correctly. Line 3 rightly reports various uses of external variables by protected objects and tasks. However, their use can be justified in each case and does not lead to race-conditions, which this rule is designed to prevent.

In the case of `gnatcheck` all the deviations described in the last paragraphs can be directly justified in the source code by using the following annotation pragma: pragma Annotate (gnatcheck, exemption_control, Rule_Name, [justification]);. This turns off the rule given by Rule_Name for a given code section [Ada13c, sec. 7.1]. `Adactl` provides a similar facility by using Ada code comments [Eur13, sec. 4.2.4].
Explanations of the `gnatcheck` rules listed in Table 3 but not mentioned in the above text can be found in [Ada13c, sec. 8]. Similarly, the `adactl` documentation covers them here [Eur13, sec. 5]. For a comparison of Ada rule checkers and the rationale behind coding rules in general refer to [Ros08].

3.2.5 Strongly Typed Programming Languages

Fulltext:
D. 49, page 115
➜ EN 50128

Ada was used for the development of the client-program and by definition of the standard that is a *strongly typed language*. The example given in Listing 2 of section 3.1.1 should provide sufficient proof by emphasizing the challenges of an intentional circumvention of the strict type checking imposed on each variable by the Ada-language. Note especially the keyword **new** (line 3) which declares a type based on Natural but is incompatible with its parent[18] and hence forces an explicit conversion (line 9) afterwards. This facility of the language which commonly results in numerous different types to accommodate all the incomparable and otherwise divergent variables,

[18] Contrary to *subtype* which would create a compatible type. – For a more detailed description refer to [Sof95, sec. 5.3.1].

	requirement	appropriate switch
1.	Coding Standard	`+RForbidden_Attributes:GNAT` (`+RForbidden_Pragmas:GNAT`)
2.	Coding Style Guide	(`+RMisnamed_Identifiers:Default`)
7.	No Unconditional Jumps	`+RGOTO_Statements` (`+RUnconditional_Exits`)
8.	Limited size and complexity of Functions, Subroutines and Methods	`+RFunction_Style_Procedures` `+RGenerics_In_Subprograms` `+RMetrics_Cyclomatic_Complexity:10` `+RMetrics_LSLOC:130` `+RMetrics_Essential_Complexity:8` `+ROverly_Nested_Control_Structures:4`
9.	Entry/Exit Point strategy for Functions, Subroutines and Methods	(`+RImproper_Returns`) `+RMultiple_Entries_In_Protected_Definitions` `+RImplicit_IN_Mode_Parameters` `+RUnassigned_OUT_Parameters` `+RParameters_Out_Of_Order`
10.	Limited use of Global Variables	(`+RNon_Visible_Exceptions`)

Table 3: Rules of `gnatcheck` to fulfill the requirements in EN 50128, table A.12

also caused `aunit`, which will be introduced in section 3.3, only to allow for simple Boolean assertion methods. Other testing frameworks of the xUnit-family usually offer such methods for each single variable type [Ada13a, footnote 1].

Contrary to Ada, the C language utilized for the server-program does not fall into the category of a *strongly typed language*. Hence, a short summary of possible measures to improve the server not only toward stronger typing but also toward a generally safer *Coding Style* (see section 3.2.4) will be given here as well.
Prior to the implementation of EN 50128, the so-called Mü 8004 by the Federal Railway Administration (EBA) was the dominant guideline for all safety-relevant machinery in the railway sector in Germany. Today, it still remains in use, but primarily in the role of a "national extension" to the internationally approved standard. Mü 8004 also encompasses a directive regarding the use of the C language. However, the available version was so outdated and vague that implementation seemed infeasible [Eis94b]. An example requirement from this paper reads: "[...] daß der Zugriff auf globale Variablen und Referenzparameter in richtiger Weise durchgeführt wird."[19] [Eis94b, point 3]. Furthermore, this and the other four (!) requirements must be [automatically] checked by an authorized [computer-]program [Eis94b, point 5]. How to define "richtige Weise" (correct way) and how to possibly check non-conforming behavior via a lint-like software, which is supposedly referred to by the second statement, is left to the reader's imagination. Apparently, more recent versions of the directive have gained significantly in size and now contain precise rules regarding which language constructs in C are actually allowed and which are not, but their meaningfulness still remains dubious at times. For example, one rule cited in a harsh critique of the directive forbids the use of non-terminating loops [Lot09, p. 14]. As this is a core

[19] Translation: "[...] that access to global variables and reference parameters will be executed in a correct way."

server-program (C)	client-program (Ada)
1. `-Wall`	1. `-fstack-check`
2. `-Wextra`	2. `-gnat05`
3. `-pedantic`	3. `-gnatwk`
4. `-Wmissing-prototypes`	4. `-gnatf`
5. `-Wfloat-equal`	5. `-gnato`
6. `-Wswitch-default`	6. `-gnatE`
7. `-Wstrict-prototypes`	
8. `-Wundef`	
9. `-std=c99`	

Table 4: `gcc`-switches used for compiling the server- and client-program

functionality of every server, no further attempts have been made to implement this new directive in any way, either.
Aside from the railway sector, there is a well-regarded *Coding Style* by the British MISRA association, targeted mainly at embedded software in the automotive sector. Since the server makes use of C99-functions like `snprintf()`, it is not compatible with the former C90-standard which the available MISRA-guidelines are based on [The98]. Newer versions incorporating the updated language specifications do exist, though. As no free tools are available to automatically check sourcecode against the guidelines and manual checking is infeasible, only a subset testable by the compiler-switches of gcc was taken into account (Table 4). The generic switches (lines 1 and 2) are based on this paper [Kar11], while other more specific ones (lines 4–8) originate from another thesis [Kai07, p. 34ff.]. `-Wunreachable-code` mentioned there had to be omitted, as it has proven to yield erroneous results [Tay11]. Similarly `-ansi` was not used since it contradicts with line 9 in the current versions of gcc which still treat the older C90 as the ANSI standard, regardless of the intermittent ratification of C99. The other checks listed in the thesis are already included in `-Wall` (line 1). For a description of the effects of the individual switches refer to the gcc manual [Sta08, ch. 3].
Despite the efforts undertaken by incorporating the compiler switches, the server-program still remains non-MISRA-compliant. For example, the usage of `<errno.h>` to check for failed conversions of `strtol()` in `protocol_ DimensionConversion()` is not allowed due to a poor definition of its behavior in the C standard [The98, rule 119]. Likewise, `<time.h>` used for determining the time of an error occurrence in `error()` (file *generic.[c|h]*) is not permitted [The98, rule 127]. However, contrary to similar[20] constraints by the Ada Ravenscar-subset (section 3.2.6), no safe alternative exists.

The Ada-switches turned on for the compilation of the client have also been shown in Table 4. However, these are mainly targeted at the conformance with the Ada-standard [Ada06] (lines 1, 5 and 6) [Ada12c, sec. 13.6] and prohibit certain programming practices (line 3), rather than enforcing safer coding styles. This is achieved by `gnatcheck` instead, which was introduced in section 3.2.4. For a full explanation of the individual switches refer to [Ada12b, sec. 3.2].

[20] but differently motivated – the expelled *Ada.Calendar*-package does allow for leap seconds and leap years which is simply not desirable for calculating time offsets in a real-time system [Ada06, sec. 9.6.1]. `<time.h>`, on the other hand, simply suffers from undefined behavior.

3.2.6 Alternative optionally required measures

There are two more optionally required measures in the standard's table mentioned in section 3.2 which are to be briefly discussed here.

Fulltext:
D. 53, page 117
➜ EN 50128

Structured Programming is targeted at component-oriented software development. By splitting up the software in small, easy to maintain parts, structural complexity should be cut down to a minimum while simplifying analysis in downstream (Figure 6) processes. This requirement will not be discussed here in detail, as complexity on a macroscopic-level (functional grouping) is enforced by the *package*-functionality of the Ada language itself (refer to the structure charts presented in Figures 10, 11, 12 and 13). On the microscopic-level, the requirement was tackled by line 8 in Table 3 of section 3.2.4.
The requirement towards the looping [CEN11, line 3381] remains incomprehensible, especially with the added "where possible". The client-program contains several loops which are completely unrelated to the input-values (because their number of iterations is obtained from *SettingsManager* for instance – see Listing 6).
Preference of structured programming over speed has been followed throughout the entire program. A counterexample would be to implement the unchecked conversion shown in Listing 2 by a much shorter address-overlay, which could then be inlined to avoid any calling overheads, like so: **for** Target'Address **use** Source'Address;
Of course, this is even more dangerous than an unchecked conversion and, on top of that, it does not physically copy the data.

Table:
A. 15, page 75
➜ EN 50128

Programming Language again just requires to stress the fact that this software is written in Ada for compliance (confer with section 3.2.5). But to add some new information here: The Ravenscar-profile intended especially for high-integrity systems has also been utilized while developing the client-software. It is composed of a set of language restrictions whose aim is a static determinability of memory usage [Ada06, sec. D. 13.1]. This has great advantages both while deploying software to memory-constrained embedded-systems as well as during possible downstream verification processes, since mathematical proofs of correctness will become much easier [BDV03, p. 7f.]. The limiting factor of the profile regarding task synchronization has already been shown in sections 3.1.5, 3.2.2 and 3.2.4. Another important measure which had to be taken care of was the banning of *Ada.Calendar* by Ravenscar. So the error-handling `Error()` in *Adaimageprocessor* had to use the compiler-specific package *GNAT.Time_Timestamp* instead to print out the time of a specific error occurrence. Furthermore, *Ada.Real_Time* was utilized to implement the stopwatch which calculates the fps-counter in *Adaimageprocessor.Output*.
For a general introduction to the Ravenscar-profile refer to [Dob10].

3.3 Unit Testing

Table:
A. 5, page 71
➜ EN 50128

Requirements towards testing fall into the "Software Component Testing Phase" (Figure 6) and are therefore no central part of this paper. However, a brief introduction to the unit-testing capabilities of Ada is given here, nonetheless. Besides a few commercial tools, two freely available test-frameworks for Ada exist: `ahven` and `aunit`. A comparison may be found here [Car11]. For the purpose of demonstration the latter has been chosen, mainly because it features a tighter integration into the development workflow of the GNAT suite aimed at university use.

The first step for larger software projects like the one developed for this paper is to run `gnattest`, a little helper program which creates a plethora of skeleton files to store the required test code [Ada13b]. A manual creation of these files is also possible but rather tedious [Ada09]. During this process each package is turned into four such skeleton files: *<Package-Name>*`-test_data.[ads|adb]` and *<Package-Name>*`-test_data-tests.[ads|adb]`. The former holds so-called *Set_Up* and *Tear_Down* routines intended for the test fixtures. These fixtures encompass everything needed to set up a test environment (e.g. create fake input values, attach hardware, set it to a certain state, etc.). The latter is intended for the actual test code. Since `aunit` is heavily influenced by the xUnit test framework, which exists for many major programming languages, this test code is based around *assertions*. These are simple Boolean statements which have to be proven right during test execution to make the test succeed.

Suppose the conversion results of `Streamconverter.ToStream()` in *Adaimageprocessor.Network*, an especially error-prone function due to its use of an unchecked conversion (see section 3.1.1), are to be tested. The specification[21] is as follows:

```
function ToStream (Input: in String) return STREAMLIB.Stream_Element_Array;
```

So a suitable test would be to feed the function with an arbitrary string, use some other mechanism to cast it into the target format (or use a static value if conversion is not feasible) and finally compare the result with what was obtained from the examined function. A possible test code for this use case is printed below:

Listing 9: Possible test code for *Adaimageprocessor.Network.Streamconverter.ToStream*

```
1  procedure Test_ToStream (Gnattest_T : in out Test) is
2     package STREAMLIB renames Ada.Streams;
3     Input : String := "Hello␣World";
4     Output : STREAMLIB.Stream_Element_Array := ToStream(Input);
5     Output_As_String : String(Input'Range);
6  begin
7     for I in Output_As_String'Range loop
8        Output_As_String(I) := Character'Val(Output(STREAMLIB.
             Stream_Element_Offset(I)));
9     end loop;
10    Assert(Input = Output_As_String, "Conversion␣yielded␣erroneous␣result."
          );
11 end Test_ToStream;
```

The actual assertion is executed in line 10 of Listing 9 with the Boolean expression to be proven as the first argument. If it evaluates to *true*, the test program compiled out of all the test cases

[21] The part which can be found in the respective file of the package whose name ends with `*.ads`

will emit a simple "PASSED" for this test. If not, "FAILED" plus the message given as the second argument will be printed out. A test may also "CRASH" if an exception occurred during execution.
To specifically check if procedures raise the exceptions they are intended to, `aunit` offers a second, dedicated assertion statement. Its application is demonstrated below for the `Send_String()` procedure in *Adaimageprocessor.Network.Socket*. Listing 11 represents the actual test code and can therefore be found in `adaimageprocessor-network-socket-test_data-tests.adb` – just analogous to Listing 9. However, since this procedure interacts with the outside world (it depends on a working socket-connection), a corresponding fixture must be set up first. This is done in Listing 10. Along with the two fixture functions discussed before, `Call_Send_String()` has been added to the set. It is a wrapper needed for technical reasons to equip `Send_String()` with adequate parameters [Ada13a, sec. 2]. In this specific case the parameter is a very long string which does not fit into a single network packet. `Send_String()` should detect this and raise an exception. Whether that actually happens is checked in line 3 of Listing 11. Only if such an exception is detected will the test be passed. Hence, this test case makes a good example of the *Error-seeding* technique mentioned in EN 50128 [CEN11, D. 21].

Listing 10: Test fixture for *Adaimageprocessor.Network.Socket.Send_String*

```
1  procedure Set_Up (Gnattest_T : in out Test) is
2  begin
3     Open_Socket("127.0.0.1", 12345);
4  end Set_Up;
5
6  procedure Tear_Down (Gnattest_T : in out Test) is
7  begin
8     Close_Socket;
9  end Tear_Down;
10
11 procedure Call_Send_String is
12    data : String (1..1000) := (others => 'X');
13 begin
14    Send_String(data);
15 end Call_Send_String;
```

Listing 11: Test code for *Adaimageprocessor.Network.Socket.Send_String*

```
1  procedure Test_Send_String (Gnattest_T : in out Test) is
2  begin
3     Assert_Exception(Call_Send_String'Access, "Exception not raised.");
4  end Test_Send_String;
```

A downside of Assert_Exception as it was presented here is its inability to check for the occurrence of a specific exception. In the case of Listing 10 and 11 only *LENGTH_EXCEPTION* is of interest. But since this exception is local to the `Send_String()`-procedure, there is no way to handle it outside (confer with line 10 in Table 3). Only for the case of library-level exceptions[22] and with a runtime which supports exception propagation such a differentiation would be possible via a tricky use of the normal Assert statement. See [Ada13a, sec. 2] for an example.

[22] exceptions defined on a more global level visible to the test code

4 OUTLOOK

The first and most important thing to mention here is the fact that the server-program has been successfully compiled for the camera (section 2.1) but until now fails to respond to any requests made via the network (section 2.2). As the same program has proven to run flawlessly on other embedded systems, it remains dubious what might be the cause of this misbehavior. Unfortunately, tracing the problem has turned out to be rather complicated since the camera's `BusyBox`-based[23] embedded Linux does come with a useless TCP-only version of `netcat`, a program to set up a very simple client- / server-architecture for network-debugging, and alternatives such as `socat` or even `tcpdump`, which is merely a network sniffer, were not readily available, either. `Gdbserver`, a network-enabled version of gdb to debug the server-program directly on the camera, is also a tool not shipped by default. Instead, it has to be enabled first by flashing the camera with a custom kernel (the heart of a Linux-system; referred to as "firmware" by Axis) whose sources are only available to accredited developers[24] [Axi07].

If eventually the server does run successfully in the future, it will remain limited to serving images to one client at a time. Depending on the practical use-case, this might not be sufficient. Via its support for *Multicasting* (a one-to-many mode of transfer) UDP would form a good starting point for this. In contrast, the TCP-based image-streams provided by the camera by default fail in the case of many simultaneous accesses due to the limited network bandwidth (see section 2.1) which is quickly used up if each client is served individually [Axi09].

The network protocol itself (section 2.2) is currently based on a static, and therefore very conservatively chosen MTU. A higher value would result in fewer packets, which in turn would mean faster transmission. However, an MTU set too highly would cause the packets to be split up by intermediate network-equipment which makes the transfer significantly slower – if it remains in a working state at all. As the MTU is network-dependent, calculating it correctly is far from a trivial task. A significant part of the TCP connection setup overhead is spent just on this matter. Switching to the new version 6 of the Internet Protocol (IPv6) could possibly provide some additional speed gains since it allows for much larger packets [BDH99]. However, such performance enhancements always come at the cost of less protection against network failures. To be precise, the time for retransmitting a lost packet becomes larger as the packet size is growing. And hence it gets more difficult to stay within the given real-time boundaries (section 1.3).

Another approach to higher network throughput is a (lossless-)compression of the image before transmission. But as this would have to be carried out in software[25] the combined time of compression, transmission and decompression would probably be higher than a plain uncompressed transfer, even for very efficient compression algorithms [Hin11, sec. 4.4].

Technically, the protocol currently suffers from the inconsistent use of NULL-terminators to mark the end of payload data in packages (compare Table 1, line 9[26] with Table 2, line 4, respectively 3). This is mainly due to the C language whose standard library-functions handle those characters very differently. However, it remains merely a cosmetic problem as Ada does not rely upon these terminating characters but rather on the length of the package itself which is encoded separately (Table 1, line 7).

[23] A collection of stripped-down versions of standard Linux-tools in a single binary.

[24] Whether this is compliant with the GPL-License of the kernel will not be discussed here.

[25] contrary to standard (lossy) compression for JPEG and H.264 which is presumably implemented in hardware on the camera

[26] "\0" stands for the NULL-terminator here

To make the protocol implementation compliant not only with the requirements of EN 50128 but also with its sister standard EN 50159 [CEN10], an encryption of the transfer may be necessary. It is often suggested to use a Message Authentication Code for this purpose [Eis94a], [FE08, ch. 4]. However, this should only be advisable for a one-to-one communication as this encryption involves a shared secret and hence becomes obsolete as soon as this secret is known to everyone.

As for the client-program, improvements could involve replacing the pointer-rich data-structures in *Adaimageprocessor.Image.Trackdata* with some more modern structures found in *Ada.Containers* [Bar06, p. 641ff.]. Moreover, the entire program could be updated by using the most recent Ada 2012 language standard instead of the currently utilized Ada 2005 version. This would allow for pre- and postconditions to functions (see section 3.1.6) [Bar13, sec. 2.3] and type invariants to provide more sophisticated variable range checking [Bar13, sec. 2.4].

The train detection algorithm (section 2.4) would significantly benefit from an accompanying track detection algorithm which could ideally allow for a *plug n' play* installation of the entire software package and hence make the "ImageTagger" introduced in section 2.3 obsolete. Several such algorithms exist: [KA09], [NHG+08], [QTS12], [Woh11].
Possibly, there are also better single-image based train detectors available which may be more robust against different weather and lighting conditions. However, no research has been conducted yet.

Finally, there is much work left to as what needs to be done regarding compliance with EN 50128. Above all, this concerns documentation. So the next steps following this paper would be to actually write both the *Software Architecture Specification* on the basis of section 3.1 [CEN11, sec. 7.3.3, p. 40] and the *Software Component Design Specification* based on section 3.2 [CEN11, sec. 7.4.3, p. 46].

5 CONCLUSION

This paper has given some insight into what has to be done to make a software system compliant with recent safety standards of the railway industry, namely EN 50128. On the basis of a given practical problem concerning the detection of a train on a video image two example programs were created (section 2), whose code was later used to exemplify selected requirements of EN 50128 (section 3). The standard turned out to pose very excessive demands on the different stages of the development process and thus it was only feasible to discuss a very small part of it in this paper. Moreover, the scattered and redundant nature of some of the measures mentioned in the standard caused various difficulties toward a clear and compelling analysis in this paper. For instance, it is certainly questionable why the underlying table of section 3.2 *highly recommends* the more demanding technique titled "Strongly Typed Programming Language" (section 3.2.5) as well as the much more general "Programming Language" 3.2.6. Likewise, the reasons for listing certain measures several times under different names can only be speculated upon (section 3.1.2).
Although the current version of the standard has only 128 pages, it certainly leaves work for

generations of students to come as it contains such a large set of all sorts of possible measures to facilitate safety, all waiting for thorough examination. However, it has to be understood there is no silver bullet to safe software. For different problems different measures apply – a fact which is also represented by the standard when demanding for a lot of different mutual inspections by the different people involved in a project. Furthermore, except for a very few cases (e.g. the *recommended measures* in the tables referred to by sections 3.1 and 3.2), the standard leaves the decision for or against a certain action to the reader. Or to put it bluntly: The standard is certainly not written like a cookbook.
As *safety* is an extremely serious issue, some more best-practice examples, perhaps in the style of this paper, would surely be helpful to prevent the inexperienced engineer from making improper decisions.

The actual coding of the software, which, contrary to what the standard expects, preceded the writing of this paper in order to make for exemplary code, was straightforward, indeed. For Ada a relatively small but exceptionally good documentation ecosystem exists. Due to its well defined and publicly available standard, there are simply not as many traps and pitfalls as with C for which an enormous amount of resources exist both on the web and in print – with a good part of it containing misleading or imperfect information. Nevertheless, the measures of the standard, especially those from section 3.2.4 involving the use of `gnatcheck` / `adactl` for the case of Ada and those from section 3.2.5 concerning the numerous gcc-switches for partial MISRA-compliance for the case of C, caused quite a few changes in the respective codebases and hence are believed to have effectively influenced the overall safety-level of the software in a positive way, even during the process of writing this paper.

In summary, the measures from EN 50128, if followed rigorously, describe a good way of increasing software-safety and help avoiding hazards originating from it. Moreover, the harmonization efforts, which must have preceded the development of this internationally approved standard, are deeply admirable – taking into account how narrow-minded and nationally-oriented the railway industry always used to be. As software development is an abstract task very much separated from the actual field of application, even a harmonization with other safety-standards of computer systems, such as those mentioned in section 1, may be conceivable in the future.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ABBREVIATIONS

ACK	Short for "acknowledged".
Ada	A programming language developed for the US Department of Defense in the 1980s. Named after Ada Lovelace.
ANSI	American National Standards Institute
API	Application Programming Interface
BMP	A lossless image format created by Microsoft for its Windows operating system.
BogoMips	A term blended from "bogus" and "mips", which is short for *million instructions per second*. Used in the Linux kernel to estimate processing speed.
C	A programming language originally developed for use in the UNIX operating system during the 1970s.
C++	A more-or-less superset of C extending it mainly with object-oriented features.
CCTV	Closed-circuit television
compiler	A computer program which translates source code into something understandable by a computer (so called *object code*). Another program, the linker, can then transform this *object code* into an executable program.
CRC	Cyclic redundancy check
EBA	Eisenbahnbundesamt
EN	European Norm
EUC	Equipment under Control, a term from ISO 61508 [Int10]
fail-safe	In the event of failure react in a way that causes no or only minimal harm to the surroundings.
FMEA	Failure mode and effects analysis
fps	frames per second
gcc	The GNU Compiler Collection
gdb	The GNU Debugger, part of the gcc
gprof	The GNU profiler, a program to conduct performance measures on other programs.
GNAT	An Ada compiler, part of the gcc
H.264	An efficient inter-frame video compression standard. Also known as MPEG-4 AVC.
Hex	Hexadecimal. A numeral system with a base of 16.
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IEC	International Electrotechnical Commission
interrupt	An external signal of interest which may come at any time and should be handled in some way by the receiver.
IP	Internet Protocol
ISO	International Organization for Standardization
JPEG	A lossy image format created by the Joint Photographics Experts Group based on Discrete Cosine Transforms and the Huffman-coding.
JSD	Jackson System Development Method
Lint	A sourcecode checker originally developed for the C language. It flags suspicious lines of code.

Linux	An operating system inspired by UNIX.
LSDM	Learmonth Structured Development Method, see [JPH07, p. 18]
MAC	Message Authentication Code, see [Int11a]
MASCOT	Modular Approach to Software Construction Operation and Test, see [JOI87]
MISRA	Motor Industry Software Reliability Association
MJPEG	A bytestream of consecutive JPEG-images.
ms	Millisecond
MTU	Maximum Transmission Unit
NIC	Network interface controller
NSA	National Security Agency
OSI	Open Systems Interconnection
PLC	Programmable Logic Controller, see [CEN11, p. 13]
PGM	Portable Graymap, a very simple image-format developed in the 1980s.
Python	A versatile, interpreted scripting language
race-condition	Describes a condition where several independent tasks (or "threads" in non-Ada-terminology) alter a common resource. Because of the concurrent nature of tasks it cannot be predicted in which order the resource is going to be accessed. This potentially leads to unwanted results.
RAM	Random-access memory, a volatile storage
Ravenscar	A subset of the Ada language for high integrity real-time software systems.
regex	A regular expression. It denotes a search pattern used predominantly for text searches in all sorts of software applications.
ROI	Region of Interest. This region forms the input pixels to any downstream image processing algorithms.
runtime	The (Ada-)Runtime is a special kind of library which can be accessed by the Ada-program through an API. It is responsible basic functionality specific to the Ada-language, the compiler and the target architecture such as range-checking and exception-propagation. For a detailed description of the runtime supplied with GNAT see [GB94].
SDK	Software Development Kit
SHA	Secure Hash Algorithm
SIL	Safety Integrity Level as defined in [CEN11].
SLOC	(physical) Source Lines of Code. Estimations have been performed using `sloccount` [Whe12].
SoC	System on a Chip, a complete computer on a single chip.
socket	An endpoint for an inter-process communication flow provided by the operating system.
SSADM	Structured Systems Analysis & Design Methodology, see [AS93]
TCP	Transmission Control Protocol
UDP	User Datagram Protocol, see [Pos80].
UYVY	a packed pixel format composed of the luminance Y and the chrominance U and V with special subsampling, see section 2.1.
UML	Unified Markup Language
XP	Extreme Programming, an agile software methodology

DECLARATION OF AUTHORSHIP

I hereby certify that the work presented here is, to the best of my knowledge, original and the result of my own investigations, except where otherwise indicated.

Dresden, July 24th, 2013

BIBLIOGRAPHY

[Ada06] ADA WORKING GROUP: *Ada Reference Manual*. `http://www.adaic.org/resources/add_content/standards/05rm/html/RM-TTL.html`. Version: 2006

[Ada09] ADACOMMONS: *AUnit Calculator Example*. `http://commons.ada.cx/AUnit_Calculator_Example`. Version: 2009

[Ada12a] ADA IN DENMARK: *catching_and_handling_interrupts_in_ada*. `http://wiki.ada-dk.org/catching_and_handling_interrupts_in_ada`. Version: 2012, last checked: 21.02.13

[Ada12b] ADACORE: *GNAT GPL User's Guide*. Document revision level 247945. 2012. – p. 528.

[Ada12c] ADACORE: *GNAT Reference Manual*. Document revision level 247883. 2012. – p. 336.

[Ada13a] ADACORE: *AUnit Cookbook*. `http://docs.adacore.com/aunit-docs/aunit.html`. Version: 2013, last checked: 14.07.13

[Ada13b] ADACORE: *GNAT Pro User's Guide: Creating Unit Tests with gnattest*. `http://docs.adacore.com/gnat-unw-docs/html/gnat_ugn_28.html`. Version: 2013, last checked: 14.07.13

[Ada13c] ADACORE: *GNATcheck Reference Manual: GNATcheck Reference Manual*. `http://docs.adacore.com/asis-docs/gnatcheck_rm.html`. Version: 2013, last checked: 11.07.13

[Ada13d] ADALOG: *AdaControl*. `http://www.adalog.fr/adacontrol2.htm`. Version: 2013, last checked: 12.07.13

[Alt11] ALTRAN PRAXIS LIMITED: *SPARK Proof Manual*. `http://docs.adacore.com/sparkdocs-docs/Proof_Manual.htm`. Version: 2011, last checked: 20.07.13

[AP13] APPELBAUM, Jacob; POITRAS, Laura: Als Zielobjekt markiert. In: *Der Spiegel* (2013), July, No. 28, pp. 22–24. – ISSN 0038–7452

[AS93] ASHWORTH, Caroline; SLATER, Laurence: *An Introduction to SSADM Version 4*. Maidenhead : McGRAW-HILL Book Company Europe, 1993. – pp. 225. – ISBN 0–07–707725–3

[Axi07] AXIS COMMUNICATIONS AB: *The Gdbserver*. `http://developer.axis.com/wiki/doku.php?id=axis:gdb-server`. Version: 2007, last checked: 22.07.13

[Axi09] AXIS COMMUNICATIONS AB: *Product performance ARTPEC-3 case*. `http://www.axis.com/files/whitepaper/wp_tech_prod_performance_artpec3_37588_en_0912_lo.pdf`. Version: 2009, last checked: 22.07.13

[Axi10] AXIS COMMUNICATIONS AB: *RAPP: RAPP User's Manual*. `http://www.nongnu.org/rapp/doc/rapp/`. Version: 2010, last checked: 08.07.13

[Axi13a] AXIS COMMUNICATIONS AB: *AXIS M3114-R Network Camera, an HDTV IP camera for buses and trains*. `http://www.axis.com/en/products/cam_m3114r/index.htm`. Version: 2013, last checked: 08.07.13

[Axi13b] AXIS COMMUNICATIONS AB: *Axis' Corridor Format - Technical guide*. `http://www.axis.com/en/products/video/about_networkvideo/corridor_format.htm`. Version: 2013, last checked: 08.07.13

[Bar06] BARNES, John: *Programming in Ada 2005*. 1st Edition. Harlow : Pearson Education, Ltd., 2006. – pp. 828. – ISBN 978–0–321–34078–8

[Bar13] BARNES, John: *Rationale for Ada 2012 Draft 7a*. http://www.ada-auth.org/standards/12rat/html/Rat12-TTL.html. Version: 2013, last checked: 22.07.13

[BDH99] BORMAN, David A.; DEERING, Stephen E. ; HINDEN, Robert M.: *RFC 2675 - IPv6 Jumbograms*. http://tools.ietf.org/html/rfc2675. Version: 1999

[BDV03] BURNS, Alan; DOBBING, Brian ; VARDANEGA, Tullio: *Guide for the use of the Ada Ravenscar Profile in high integrity systems*. http://www.sigada.org/ada_letters/jun2004/ravenscar_article.pdf. Version: 2003

[Bor97] BORN, Günter: Die PBM-Formate (PBM, PGM, PPM). In: *Referenzhandbuch Dateiformate*. 5. Auflage. Bonn : Addison-Wesley Publishing Company, 1997. – ISBN 3–8273–1241–8, Kapitel 62, pp. 1068–1070

[BPB88] BORMAN, D.; PARTRIDGE, C. ; BRADEN, R.: *RFC 1071 - Computing the Internet checksum*. http://tools.ietf.org/html/rfc1071. Version: 1988, last checked: 12.07.13

[Bud11] BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS: *Software in safety - critical systems*. http://www.inf.mit.bme.hu/sites/default/files/materials/category/kateg%C3%B3ria/oktat%C3%A1s/v%C3%A1laszthat%C3%B3-t%C3%A1rgyak/kritikus-be%C3%A1gyazott-rendszerek/11/KBR-2011_EA04a_Software.pdf. Version: 2011, last checked: 18.07.13

[Car11] CARREZ, Stephane: *Aunit vs Ahven*. http://blog.vacs.fr/index.php?post/2011/11/27/Aunit-vs-Ahven. Version: November 2011

[CEN10] CENELEC: *EN 50159 - Railway applications - Communication, signalling and processing systems - Safety-related communication in transmission systems*. 2010

[CEN11] CENELEC: *EN 50128 - Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*. 2011

[Dan10] DANG, Bruce: *CCC-TV - Adventures in analyzing Stuxnet*. http://media.ccc.de/browse/congress/2010/27c3-4245-en-adventures_in_analyzing_stuxnet.html. Version: 2010, last checked: 13.07.13

[Dob10] DOBBING, Brian: The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In: *Ada User Journal* 31 (2010), No. 1, 47–54. http://www.ada-europe.org/archive/auj/auj-31-1.pdf

[Dor06] DORST, Wim van: *BogoMips mini-Howto*. http://tldp.org/HOWTO/BogoMips/. Version: 2006, last checked: 10.07.13

[Eis94a] EISENBAHNBUNDESAMT: *Informationstechnik - Sicherheitstechnik - Datensicherheitsmechanismus mit Chiffriercheck unter Anwendung eines Algorithmus zur Ver- und Enschlüsselung von Datenblöcken*. Bonn, 1994

[Eis94b] EISENBAHNBUNDESAMT: *Mü 8004 - Leitlinie mit ergänzenden Bestimmungen für die Prüfung der Software in C*. Bonn, 1994

[Eur13] EUROCONTROL/ADALOG: *AdaControl User Guide V1.15r5*. http://www.adalog.fr/compo/adacontrol_ug.html. Version: 2013, last checked: 12.07.13

[FE08] FAIRHURST, Godred; EGGERT, Lars: RFC 5405 - Unicast UDP Usage Guidelines for Application Designers. (2008). http://tools.ietf.org/html/rfc5405

[GB94] GIERING, E. W.; BAKER, T. P: The GNU Ada runtime library (GNARL). In: *Proceedings of the eleventh annual Washington Ada symposium & summer ACM SIGAda meeting on Ada - WADAS '94*. New York, New York, USA : ACM Press, July 1994. – ISBN 0897916840, 97–107

[Hin11] HINTERMAIER, Wolfgang: *An IP-based System Architecture for camera-based Driver Assistance Services*, TU München, Dissertation, 2011. – p. 139.

[Int10] INTERNATIONAL ELECTROTECHNICAL COMMISSION: *IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related system*. 2010

[Int11a] INTERNATIONAL ELECTROTECHNICAL COMMISSION: *IEC 9797 - Information technology - Security techniques - Message Authentication Codes (MACs)*. 2011

[Int11b] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO 26262 - Road vehicles. Functional safety.* 2011

[JFCS03] JACOBSON, V.; FREDERICK, R.; CASNER, S. ; SCHULZRINNE, H.: RTP: A Transport Protocol for Real-Time Applications. (2003). `http://tools.ietf.org/html/rfc3550`

[JOI87] JOINT IECCA & MUF COMMITTEE ON MASCOT: *THE OFFICIAL HANDBOOK OF MASCOT*. Version 3. Malvern : Royal Signals and Radar Establishment, 1987. – p. 324. `http://async.org.uk/Hugo.Simpson/MASCOT-3.1-Manual-June-1987.pdf`

[JPH07] JOHNSON, Brian; PUBLISHING, Van H. ; HIGGINS, John: *Itil (R) and the Software Lifecycle: Practical Strategy and Design Principles*. Van Haren Publishing, 2007. – pp. 122. – ISBN 9087530498

[KA09] KALELI, Fatih; AKGUL, Yusuf S.: Vision-based railroad track extraction using dynamic programming. In: *2009 12th International IEEE Conference on Intelligent Transportation Systems* (2009), October, 1–6. `http://dx.doi.org/10.1109/ITSC.2009.5309526`. – DOI 10.1109/ITSC.2009.5309526. ISBN 978–1–4244–5519–5

[Kai07] KAISER, Guido: *Exemplarisches Eclipse-Plugin zur statischen C-Syntaxanalyse am Beispiel von Misra-C Regeln*, Hochschule für Angewandte Wissenschaften Hamburg, bachelor thesis, 2007. `http://opus.haw-hamburg.de/volltexte/2007/354/pdf/Bachelorarbeit_Final.pdf`

[Kar11] KAREL DE GROTE-HOGESCHOOL TERA LABS: *Coverage of MISRA - C 2004 by gcc warnings*. `http://www.iwt-kdg.be/teralabs/sites/default/files/SomenotesongccwarningsV3.pdf`. Version: 2011, last checked: 05.07.13

[Lan10] LANGNER, Ralph: *The short path from cyber missiles to dirty digital bombs*. `http://www.langner.com/en/2010/12/26/the-short-path-from-cyber-missiles-to-dirty-digital-bombs/`. Version: 2010, last checked: 17.07.13

[Lot09] LOTTERMOSER, Martin: *Mängel der Programmierregeln des Eisenbahn-Bundesamtes*. `http://home.htp-tel.de/lottermose2/Maengel-EBA.pdf`. Version: 2009

[NHG+08] `http://homepages.ucalgary.ca/~gjhay/geobia/GEOBIAPaperslinkedPDF/6718_Neubert_Proc_Pap.pdf`

[PJ95] PAGE-JONES, Meilir: *Strukturiertes Systemdesign - ein praktischer Leitfaden*. München : Hanser, 1995. – pp. 378. – ISBN 3–446–16302–6

[Pos80] POSTEL, J.: *RFC 768 - User Datagram Protocol*. `http://tools.ietf.org/pdf/rfc768.pdf`. Version: 1980

[Pos86] POST, Julian: Application of a structured methodology to real-time industrial software development. In: *Software Engineering Journal* 1 (1986), No. 6, pp. 222–235. `http://dx.doi.org/10.1049/sej:19860035`. – DOI 10.1049/sej:19860035. – ISSN 0268–6961

[QTS12] QI, Zhiquan; TIAN, Yingjie ; SHI, Yong: Efficient railway tracks detection and turnouts recognition method using HOG features. In: *Neural Computing and Applications* (2012), February. `http://dx.doi.org/10.1007/s00521-012-0846-0`. – DOI 10.1007/s00521–012–0846–0. – ISSN 0941–0643

[Red06] REDER, H.-J.: Cross-acceptance of safety approvals in the rail industry: a manufacturer's viewpoint. In: *1st IET International Conference on System Safety* vol. 2006, IEE, 2006. – ISBN 0 86341 646 2, 338–343

[Ros08] ROSEN, J.-P.: A comparison of industrial coding rules. In: *Ada User Journal* 29 (2008), No. 4, 277–281. http://www.ada-europe.org/archive/auj/auj-29-4.pdf

[RTC92] RTCA INC.: *Software Considerations in Airborne Systems and Equipment Certification*. http://www.rtca.org/store_product.asp?prodid=581. Version: 1992

[Sch05] SCHMITT, E.: *Die Varianten des YUV Komponenten-Signals*. http://www.cine4home.de/knowhow/ChromaUpsampling/ChromaUpsampling.htm. Version: 2005, last checked: 03.04.13

[Sch09] SCHIEL, James: *Enterprise-Scale Agile Software Development*. Hoboken : CRC Press, 2009. – ISBN 9781439803226

[sec13] SECUNET SECURITY NETWORKS AG: *Anet*. http://www.codelabs.ch/anet/index.html. Version: 2013, last checked: 21.07.13

[SFR03] STEVENS, W. R.; FENNER, Bill ; RUDOFF, Andrew M.: *The Sockets Networking API: UNIX® Network Programming Volume 1*. 3. Edition. Boston, MA : Addison-Wesley Professional, 2003. – pp. 993. – ISBN 0–13–141155–1

[SK12] SCHÜTTE, Jörg; KAISER, Hans-Christian: Ein neues Verfahren zur automatisierten Erkennung von Hindernissen im Bahnsteiggleis. In: *Signal+Draht* 104 (2012), No. 12, pp. 15–20

[Sof95] SOFTWARE PRODUCTIVITY CONSORTIUM: *Ada 95 Quality and Style Guide: Guidelines for Professional Programmers*. http://www.adaic.org/resources/add_content/docs/95style/html/cover.html. Version: 1995, last checked: 12.07.13

[SPMR06] STIGGE, Martin; PLÖTZ, Henryk; MÜLLER, Wolf ; REDLICH, Jens-Peter: *Reversing CRC - Theory and Practice*. http://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2006-05/SAR-PR-2006-05_.pdf. Version: 2006

[Sta08] STALLMAN, Richard M.: *Using the GNU Compiler Collection*. For GCC Version 4.5.4. 2008. – p. 702.

[Tay11] TAYLOR, Ian L.: *Re: gcc -Wunreachable-code option*. http://gcc.gnu.org/ml/gcc-help/2011-05/msg00360.html. Version: 2011, last checked: 19.07.13

[Tec05] TECHNOLOGY SYSTEMS: *NetBSD Toaster with the TS-7200 ARM9 SBC*. http://www.embeddedarm.com/software/arm-netbsd-toaster.php. Version: 2005, last checked: 09.07.13

[The98] THE MOTOR INDUSTRY RESEARCH ASSOCIATION: Guidelines For The Use Of The C Language In Vehicle Based Software. (1998), No. July

[Val11] VALURE, Greg: *Natural Docs*. http://www.naturaldocs.org/. Version: 2011, last checked: 12.07.13

[Whe12] WHEELER, David A.: *SLOCCount*. http://www.dwheeler.com/sloccount/. Version: 2012, last checked: 19.07.13

[Wil11] WILSON, Dave: *YUV pixel formats*. http://www.fourcc.org/yuv.php#UYVY. Version: 2011, last checked: 03.04.13

[WM85] WARD, Paul T.; MELLOR, Stephen J.: *Structured Development for Real-Time Systems - Volume 2: Essential Modeling Techniques*. Englewood Cliffs, N.J. : Prentice Hall, 1985. – pp. 163. – ISBN 0–13–854795–5

[Woh11] WOHLFEIL, Jürgen: Vision based rail track and switch recognition for self-localization of trains in a rail network. In: *2011 IEEE Intelligent Vehicles Symposium (IV)* (2011), June, No. Iv, 1025–1030. http://dx.doi.org/10.1109/IVS.2011.5940466. – DOI 10.1109/IVS.2011.5940466. ISBN 978–1–4577–0890–9

[YC79] YOURDON, Edward; CONSTANTINE, Larry L.: *Structured Design - Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs, N.J. : Prentice Hall, 1979. – pp. 471. – ISBN 0–13–854471–9