Assignment of bachelor's thesis

Title:	Image data analysis on embedded systems
Student:	Michal Pech
Supervisor:	Ing. Miroslav Macík, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering 2021
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

The VENT-CONNECT [1,2] system, developed at the Czech Technical University in Prague, is a remote surveillance tool for monitoring instruments in intensive care, including lung ventilators and vital signs monitors. Currently operational at Kralovske Vinohrady University Hospital, the system allows remote access to on-screen images of ventilators and other medical instruments. The current focus is on analyzing curves and other information utilizing VENT-CONNECT units - special devices with HDMI data input based on a microcomputer. In this context, the development of the VentVision module aims to utilize the C programming language and the GStreamer library for efficient parsing and subsequent analysis of data obtained from the ventilator screen.

Student will focus on the following tasks:

Examine the current implementation and development of the VENT-CONNECT system, focusing on modules related to the VentVision component. Investigate the Anonfilter module, which directly precedes VentVision, and analyze their interconnections. Understand the functionalities of the GStreamer library.
Identify optimal ways to leverage GStreamer for analyzing image data from lung ventilator screens. Focus on extraction and analysis of curves and other relevant data (alphanumerical values, alarms, etc.)
Develop the VentVision module emphasizing parsing and analyzing data obtained from the ventilator screen. Ensure seamless integration with the existing Anonfilter module and efficient utilization of the GStreamer library.

Electronically approved by Ing. Michal Valenta, Ph.D. on 5 December 2023 in Prague.

Evaluate the VentVision module capabilities of exporting analyzed data to facilitate its utilization by other components within the VENT-CONNECT system.
Integrate the finalized VentVision module into the VENT-CONNECT system, ensuring compatibility with the Anonfilter module and establishing a reliable mechanism for exporting parsed and analyzed data for further system utilization.

Literature:
[1] https://ventconnect.cz, ONLINE, accessed 2022-10-01
[2] Vysloužilová et al. Nová technologie v intenzivní péči pomáhá vzdáleně sledovat pacienty nejen s COVID-19, Medsoft 2021, available form: https://creativeconnections.cz/medsoft/2021/Medsoft_2021_Vyslouzilova.pdf

Electronically approved by Ing. Michal Valenta, Ph.D. on 5 December 2023 in Prague.

Bachelor’s thesis

IMAGE DATA ANALYSIS ON EMBEDDED SYSTEMS

Michal Pech

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Miroslav Macík, Ph.D.
May 16, 2024

Citation of this thesis: Pech Michal. *Image data analysis on embedded systems.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

I would like to express my sincere gratitude to my thesis supervisor, Ing. Miroslav Macík, Ph.D., for his invaluable guidance throughout my bachelor's thesis. I would also like to extend special thanks to Ing. Jan Jirman and Ing. Milan Němý, Ph.D., for their priceless assistance with my work. Additionally, my heartfelt appreciation goes to the entire VENT-CONNECT team.
Above all, I am profoundly grateful to Jesus Christ, my Lord and Savior, who is the purpose of my life and, consequently, of this thesis.

Declaration

In Prague on May 16, 2024

Abstract

This bachelor thesis focuses on the development of the VentVision module. As part of the VENT-CONNECT system for monitoring lung ventilation at the Kralovske Vinohrady University Hospital, which was developed at the Czech Technical University in Prague, VentVision solves the problem of parsing data from the lung ventilator screen, including curves and numerical data that are relevant for the hospital personnel.

VentVision processes data using the C programming language and the GStreamer library. The data parsing itself uses a function that substitutes OCR (Optical Character Recognition) and works by comparing specific locations in the image with stored templates. This approach proved to be more reliable in this particular project than using any of the open source OCR engines. The solution was compared against a similar solution within the project and also demonstrated through a test in the hospital that it was capable of continuously processing ventilator data over an extended period of time.

The result of the work is a functional module that reliably digitises data from the lung ventilator screen. The benefit of this work is to help hospital personnel respond more quickly to the needs of patients in the intensive care unit, significantly increasing the hospital's preparedness and efficiency in managing everyday operations but also making it easier to deal with a potential crisis such as the COVID-19 pandemic.

Keywords curve parsing, digitalisation, Hamilton ventilator, VENT-CONNECT, OCR, C, GStreamer

Abstrakt

Tato bakalářská práce se zabývá vývojem modulu VentVision. Modul VentVision, který je součástí systému VENT-CONNECT pro monitorování plicní ventilace ve Fakultní nemocnici Královské Vinohrady a který byl vyvinut na ČVUT v Praze, řeší problematiku zpracování dat z obrazovky plicního ventilátoru, včetně křivek a číselných údajů, které jsou významné pro personál nemocnice.

VentVision zpracovává data pomocí programovacího jazyka C a knihovny GStreamer. Samotné parsování dat využívá funkci, která nahrazuje OCR (Optical Character Recognition) a funguje na základě porovnávání konkrétních míst v obraze s uloženými šablonami. Tento přístup se v tomto konkrétním projektu ukázal jako spolehlivější než použití některého z open source OCR enginů. Řešení bylo porovnáváno s podobným řešením v rámci projektu a také bylo prostřednictvím testu v nemocnici prokázáno, že je schopno nepřetržitě zpracovávat data z ventilátoru po delší dobu.

Výsledkem práce je funkční modul, který spolehlivě digitalizuje data z obrazovky plicního ventilátoru. Přínos této práce spočívá v tom, že pomáhá nemocničnímu personálu rychleji reagovat na potřeby pacientů na jednotce intenzivní péče, což významně zvyšuje připravenost a efektivitu nemocnice při zvládání každodenního provozu, ale také usnadňuje řešení potenciálních krizí, jako byla pandemie COVID-19.

Klíčová slova parsování křivek, digitalizace, ventilátor Hamilton, VENT-CONNECT, OCR, C, GStreamer

List of abbreviations

API	Application Programming Interface
LSD	Left Side Digits
OCR	Optical Character Recognition
PCC	Pearson Correlation Coefficient
RPI	Raspberry Pi
RSD	Right Side Digits

Introduction

Over the past four years, hospitals worldwide have faced significant challenges due to the COVID-19 pandemic [1]. A significant percentage of COVID-19 patients have trouble breathing and need help from ventilators [2].

Vanek and Macik [3] describe the problems faced by doctors at the Kralovske Vinohrady University Hospital. In particular, there are only a limited number of doctors who can properly adjust ventilation, and access to patients has been made more difficult by the need for a full-body suit during the most serious pandemic. So he and the rest of the VENT-CONNECT team decided to create the VENT-CONNECT system.

The VENT-CONNECT system, developed at the Czech Technical University in Prague, is a telemedicine application that allows remote monitoring of lung ventilator screens and vital signs monitors whose data outputs are not directly connected to the hospital information system [4].

The subject of this thesis is the VentVision module. It should function on VENT-CONNECT units, which are devices connected to the HDMI data input from the ventilators. Its job is to analyze, parse and export the visual data from the ventilator screen while the ventilator is running so that the data can be further processed by subsequent parts of the system. It will utilize the C programming language and the GStreamer library for this purpose.

An effective solution to this task could help hospital staff respond more quickly to potential ventilation problems and prevent further problems by optimising ventilation settings based on the results of ventilator data analysis.

Goals

The main goal of this thesis is to create the VentVision module, mentioned above, for the GStreamer library, that will parse relevant data from the screen like curves, and connect the module to the rest of the VENT-CONNECT system.

To complete these tasks the following will be needed to accomplish:

1. Examine the current implementation and development of the VENT-CONNECT system, focusing on modules related to the VentVision component. Investigate the AnonFilter module, which directly precedes VentVision, and analyze their interconnections. Describe the functionalities of the GStreamer library.

2. Identify optimal ways to leverage GStreamer for analyzing image data from lung ventilator screens. Focus on extraction and analysis of curves and other relevant data (alphanumerical values, alarms, etc.).

3. Develop the VentVision module emphasizing parsing and analyzing data obtained from the ventilator screen. Ensure seamless integration with the existing AnonFilter module and

efficient utilization of the GStreamer library.

4. Evaluate the VentVision module capabilities of exporting analyzed data to facilitate its utilization by other components within the VENT-CONNECT system.
5. Integrate the finalized VentVision module into the VENT-CONNECT system, ensuring compatibility with the AnonFilter module and establishing a reliable mechanism for exporting parsed and analyzed data for further system utilization.

In conclusion, the development of the VentVision module represents a major step forward in extending the capabilities of the VENT-CONNECT system for remote monitoring and analysis of vital ventilator data. By leveraging the capabilities of the GStreamer library, this module aims to provide hospital staff with a real-time overview of ventilation parameters, allowing them to quickly respond to potential problems and optimize ventilation settings. The successful implementation and integration of VentVision hold promise for improving patient care and outcomes, especially in the face of the challenges presented by the COVID-19 pandemic. This work therefore seeks to contribute to ongoing efforts to strengthen healthcare infrastructure and resilience in the face of unprecedented crises.

Chapter 1

Analysis

In this chapter, we analyze the project and solutions to similar problems. First of all, we must describe the current state of the whole domain, into which our VentVision module will be integrated.

1.1 Domain analysis

The data we are looking at come from a ventilator in the intensive care unit at the Kralovske Vinohrady University Hospital. This data is transmitted as a video signal to the VENT-CONNECT system, which includes the VentVision module. The VENT-CONNECT system already contains a solution to the problem we are dealing with. However, this solution operates on an offline basis on the developer's computer and thus only works with the recorded records, which the VentVision module uses only as test data. The VentVision module itself will run on a Raspberry Pi 4 device in the hospital. We will now take a closer look at each of these parts of the hospital system.

1.1.1 Ventilation

Our module is being developed for the Hamilton G5 [5] lung ventilator. Although in the near future we are interested in extending it to work with other intensive care devices, for now we will only focus on this ventilator.

"The HAMILTON-G5 ventilator is designed for intensive care ventilation of adult and pediatric patients, and optionally infant and neonatal patients. The device is intended for use in the hospital and institutional environment where health care professionals provide patient care. The HAMILTON-G5 ventilator is intended for use by properly trained personnel under the direct supervision of a licensed physician." [6, p. 7] Since the hospital only has a limited number of these properly trained personnel to work with ventilators, the goal of our system is to help them more quickly assess the ventilation status and more easily find the optimal ventilator settings.

The ventilator screen is LCD type and the image resolution is 1024 x 768 pixels [6, p. 381]. This resolution is essential for our work with image data. Since most of the data we are looking at is in fixed locations on the screen, we can easily find it by the position given by the pixel number. In the Figure 2.1 you can find an example of what a single frame from the ventilator screen looks like. Video signal is output via DVI-I connector [6, p. 54]. The frame rate of the video signal is 10 frames/second.

1.1.2 VENT-CONNECT

In the Figure 1.1 you can see the architecture of the VENT-CONNECT system, which consists of multiple VENT-UNIT devices, Data server and OctoConnect [7].

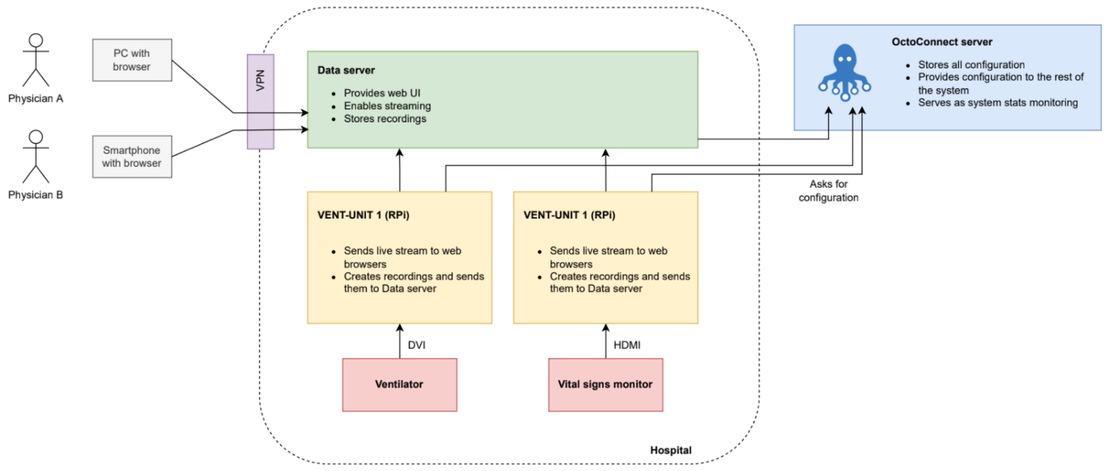


Figure 1.1 The architecture of the VENT-CONNECT system by Jirman [7]

VENT-UNIT devices can be connected to various intensive care devices - these can be the aforementioned Hamilton ventilators, which our module also works with, or vital signs monitors. Currently, their primary task is to send a live stream to a web browser. Vanek [3] further states that at the same time, in addition to its normal operation, the device records a 10-second video every 15 minutes. This video is then uploaded to the Data server using an SSH (secure shell) based protocol. This process was used as a substitute for live streaming of the screen in earlier version of the system, as live-streaming support was challenging to implement. Now we use these recordings as test data to develop our module.

The Data server provides web UI for authorized hospital personnel and system developers. As mentioned, in this version of the system, it enables the monitoring of the live stream from the intensive care devices' screens as well as the past recordings, which the Data server stores.

OctoConnect is a system that interconnects all devices and servers in the VENT-CONNECT system. It is an administrative dashboard to which all the devices and data servers are connecting and asking it for configuration. [7]

1.1.3 Similar solution in the project

Ing. Milan Nemy, Ph.D. has developed a similar solution, which is part of the learning phase of the development of the classifier, which will run on the server and should in its final form continuously evaluate the data that will be obtained from our VentVision module to determine the alarm level and propose the optimal ventilation settings.

This solution implements an algorithm for parsing the curves from the ventilator screen in MATLAB [8] consists of the following steps:

1. The screen image is cropped according to precise coordinates to contain only curves.
2. The colors of the image are quantized to a few colors that can be easily distinguished from each other. This step is necessary because the algorithm works with compressed video recordings.
3. Detection of the fully loaded curve is performed. This requires going through the cropped portion of the screen and checking if there is any gap in the curve to indicate the presence

of a running cursor. To ensure that there is no gap in the curve, the legend in the upper left corner must first be filtered out, as it is the same color as the curve and could easily be mistaken for a curve that has the cursor just mentioned at that point. You can see the running cursor and the legend example shown in the Figure 1.2.

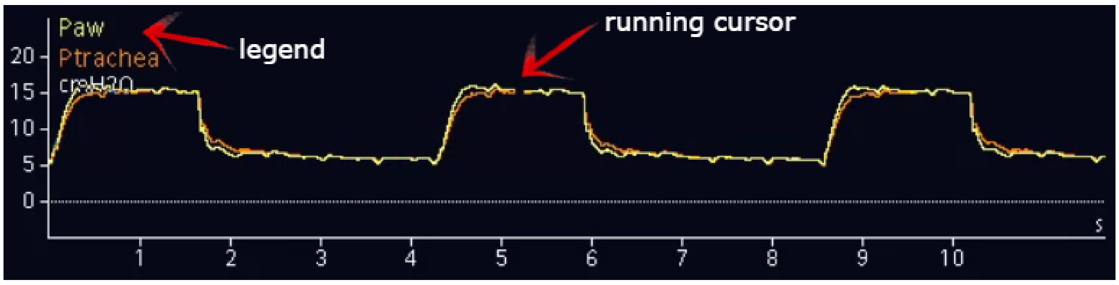


Figure 1.2 Running cursor and legend shown in the Paw curve example

4. Once the entire curve is loaded, the Paw and Flow curves are parsed into time series. To do this, the scales of the X and Y axes need to be estimated. Again, according to the precisely determined coordinates, the ticks on the axis must be found to locate a given numerical label. Signal extraction is then performed by thresholding the corresponding colour component. Short breaks can appear in the signal (1-2 pixels). They can be repaired, for example, by linear interpolation.

As can be seen from the description, the solution does not deal with the case that the curves are not synchronized and therefore not both fully loaded at the same time. However, this situation can occur in exceptional cases. This solution was a great inspiration for our final solution to the problem, as you can see in the implementation section below. Nevertheless, our solution is expected to handle exceptional cases, such as the one we just mentioned, on which this algorithm fails.

1.1.4 Target platform (RPI)

The so-called VENT-UNIT devices, which mediate the transmission of the video signal to the server from the hospital and on which, among other things, our VentVision module works, are Raspberry Pi 4 Model B [9] microcomputers running a special image created specifically for this purpose [7].

In the Figure 1.3 you can see how the video output of the ventilator is connected to the VENT-UNIT and also how the VENT-UNIT accesses the internet. According to internal documentation of VENT-CONNECT, in case the communication with the internet and therefore the connection to the Data server fails, VENT-UNIT, according to its basic configuration, switches to a mode where it stores the read data in its own hard disk so that it does not lose it, and transfers it to the server at the earliest opportunity.

The Raspberry Pi 4 Model B features a high-performance 64-bit quad-core processor, hardware video decode at up to 4Kp60, up to 8GB of RAM, dual-band 2.4/5.0 GHz wireless LAN, Gigabit Ethernet and more [9]. Thanks to the fact that the system runs on a special custom image by VENT-CONNECT team, it would be possible to switch to another microcomputer with similar capabilities if necessary.

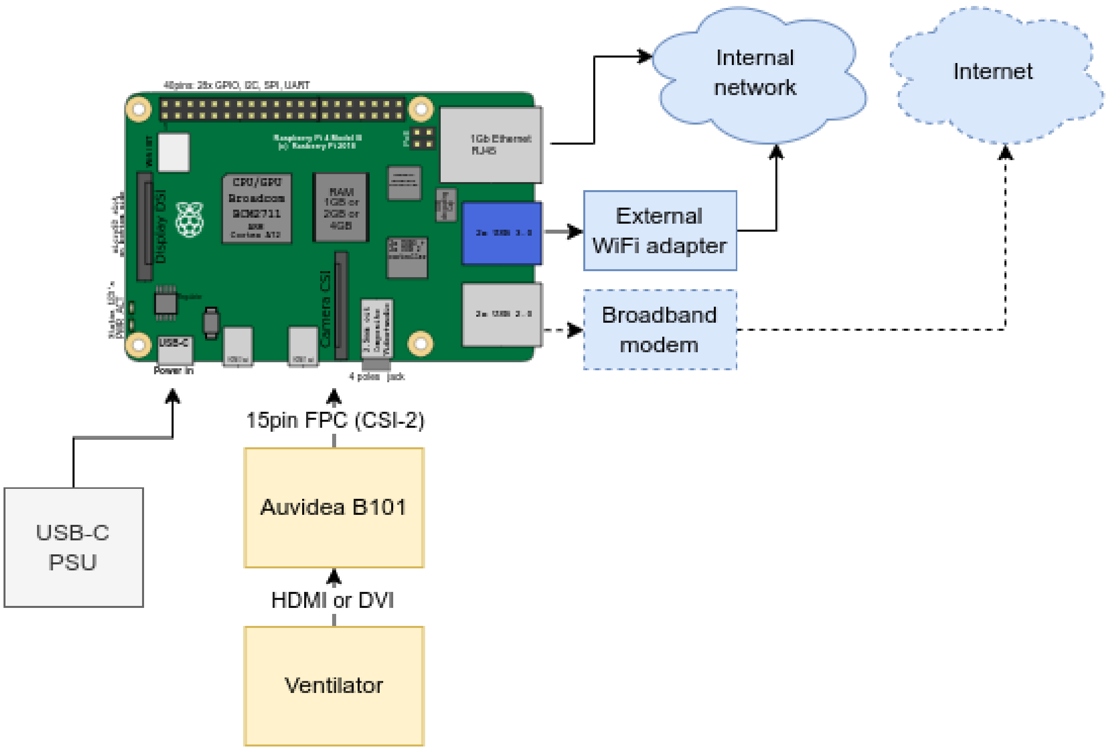


Figure 1.3 Raspberry Pi 4 integration diagram from internal VENT-CONNECT documentation

1.2 State of the art

Parsing data from the ventilator screen consists of two parts, which are also areas that need to be explored simultaneously, namely OCR and so-called line chart parsing.

1.2.1 OCR

Optical Character Recognition (OCR) is the process of electronically converting images containing typed, handwritten, or printed text into machine-readable text. OCR finds applications across diverse fields, such as digitizing data from printed records, forms, and documents, as well as recognizing text on road signs, among others. The key benefits of employing OCR technology include its ability to transform images into searchable, editable, and easily storable electronic formats. [10] This thesis will primarily concentrate on swiftly identifying axis labels and relevant values, which are consistently positioned at specific coordinates within the frame of video stream. For this purpose, only OCR engines compatible with C/C++ are to be utilized.

GOCR [11] is a free OCR engine, developed under the GNU Public License. It is a simple and fast engine that requires no training data. Its recognition process consists of two passes. In the first pass, the entire document is processed. In the second pass, unknown characters are processed. [12] According to the results of Dhiman's study [12], it seems that GOCR could return reliable results in our case, as evidenced by my experiments with the ventilator screenshots, in which GOCR was able to reliably recognize most of the characters, except for the confusion of the character "0" with "o" and the error in recognizing the Czech language accented characters. However, despite GOCR's ability to handle a variety of image formats, in our case we would first have to convert the image from the format passed by the GStreamer pipeline to one of the supported formats.

Tesseract [13] is an open-source OCR engine developed in C++ that allows developers to use C or C++ API to develop their own applications. The architecture of Tesseract involves several key steps. First, input in the form of grayscale or colored images is provided to the engine. These images are then converted into binary images using adaptive thresholding. Next, connected component analysis is performed to store the outlines of components, which are then grouped into blobs. These blobs are organized into text lines and further segmented into words based on the spacing between characters, utilizing fuzzy spaces. Recognition occurs through a two-pass process: in the first pass, words are recognized and passed to an adaptive classifier for training data, while in the second pass, unrecognized words from the first pass are recognized. Finally, fuzzy spaces are resolved in the final phase, resulting in the extraction of text from the image. [14] [12] This approach does not solve our problem optimally, as we explain in the Analysis conclusion section.

1.2.2 Waveform parsing

The main problem that our work addresses is loading curves (waveforms) and converting them to time series. A seemingly similar problem has been addressed in the past by papers calling the problem line graph parsing [15]. Unfortunately, their solutions are not suitable for us, since these line charts are defined as graphs with points that are connected by straight lines. In order to apply such a definition to the curves that we plot, we would have to find such a point on each x-coordinate of the graph. Therefore, this solution is too complex for our problem.

Although we cannot take direct inspiration from the curve parsing algorithm, we will at least use in our considerations the formula for calculating the scale of the X and Y axes given by Kumar [16] in his paper. To find these scales, he divides the average of the actual x-label or y-label ticks (N_{ticks}) by the average distance between ticks in pixels (Δd).

$$\alpha = \frac{N_{\text{ticks}}}{\Delta d}$$

In our case it will not even be necessary to calculate the mentioned average, but more about this in the Implementation chapter.

1.3 Analysis conclusion

In this chapter, we have discussed the current status of the VENT-CONNECT system into which our module will be integrated. This will be discussed in detail in the next chapter, which will describe how our module works. The description of a similar solution in this chapter will serve partly as a guide on which we will lightly rely. For now, the parameters of the device on which our module will run give us at least a vague idea of how efficient our module must be.

Next, we discussed OCR. In our project, it is not necessary to use a conventional OCR engine to look for the positions where individual letters and digits are located, as most of these values are fixed within the layout of the ventilator screen. In addition, the data on the screen can be compared against a limited set of pre-known values, since the ventilator shows either numerical data consisting of digits or labels of the values it measures. Thus, while our solution resembles the working of OCR, it is a simpler variation of it that provides all the functionality we need without unnecessary complexity at the required speed. In this solution we take inspiration from work by Shamir [17] and use the Dice similarity coefficient, in our case a simplified version of it. Here we first give the formula of the classical Dice coefficient:

$$Dice(A, B) = \frac{2 \times |A \cap B|}{|A| + |B|}$$

.

In our work we always compare two images of the same size, therefore $|A| = |B|$ which means $|A| + |B| = 2|A|$ and the the fraction in the Dice coefficient can be shortened like this:

$$Dice(A, B) = \frac{|A \cap B|}{|A|}$$

.

Finally, we come to the actual problem of parsing curves from the ventilator screen. Past work did not directly help us in the process of how to parse a curve, but we did at least take away some inspiration about parsing information from graphs. Specifically, the method of computing the scale of the axes of a graph.

Chapter 2

Design

This chapter deals with the design process of creating the module and its integration into the VENT-CONNECT system. Before it is time to delve into the intricacies of system architecture and integration, it is first necessary to address functional and non-functional requirements.

2.1 Functional requirements

In this section, the functional requirements for the GStreamer module will be articulated, depicting the specific actions it must perform to fulfill its designated role as a filter for medical ventilator video streams. These requirements include frame analysis, curve detection, data extraction and signal emission among others.

F1. Frame analysis

Because the layout and many other values in the stream may differ from frame to frame, it is essential for the proper functioning of the module that it analyzes every single frame. This analysis should consist of checking the correct ventilator layout (i.e. Hamilton) and detecting, whether the data on the screen are loaded and therefore can be parsed and exported. In 2.1 you can see an example of a (slightly altered) frame.

F2. Curve detection

The module must employ algorithms to detect curves displayed on the screen within each frame. Is focuses mainly on Paw and Flow (which can be found labeled as *Průtok* in 2.1) curves. Due to the healthcare environment, there is a strong emphasis on accuracy and therefore it is imperative that excessive rounding is avoided.

F3. Numerical values detection

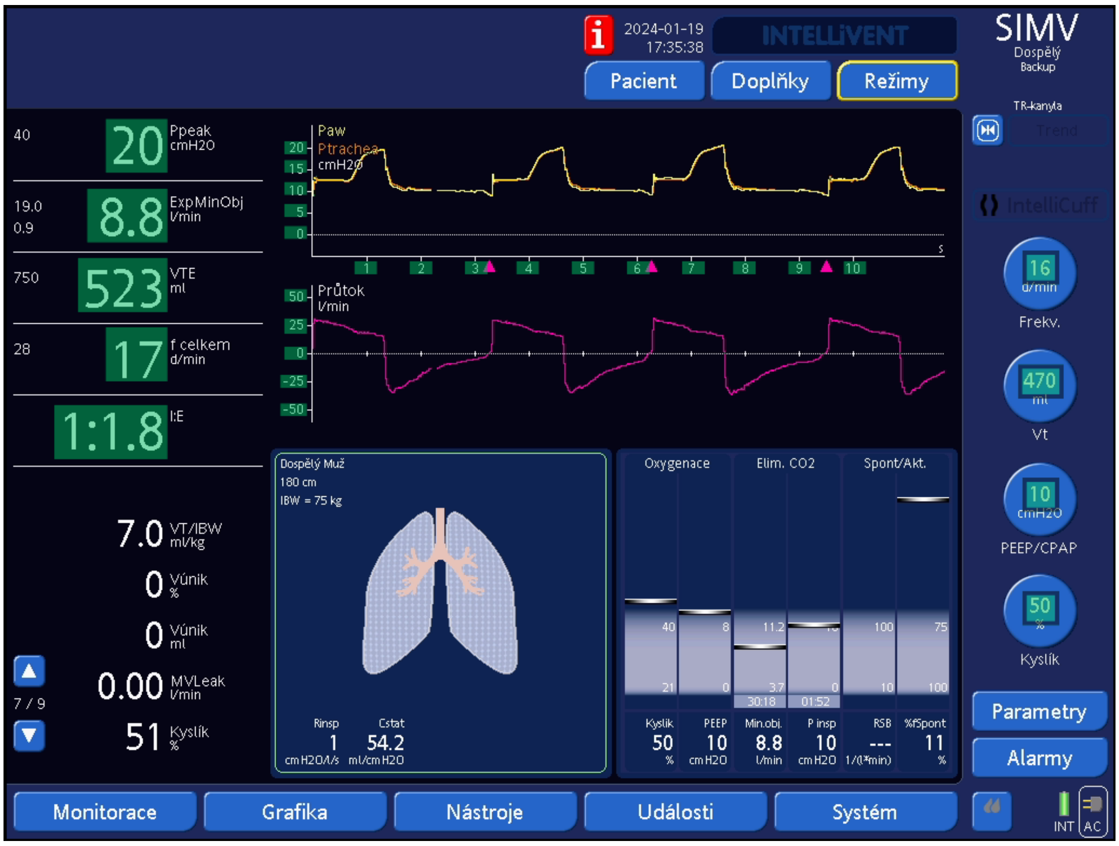
In addition to the curves, there are numerical values on the lung ventilator screen that are relevant for data analysis. The values that are to be recognised by the module are marked in 2.1.

F4. Signal emission

The data that we recognized in the image needs to be sent to the server. To do this, it is desirable to use the GStreamer module's feature of emitting a signal.

F5. Integration with GstAnonymizerModule

The module must integrate seamlessly with the GstAnonymizerModule, which is already integrated in the system and anonymizes personal data of the patient on the frame. All data processed and transmitted by the VentVision module should be anonymized to adhere to privacy regulations and ensure patient confidentiality.



■ **Figure 2.1** Figure of Hamilton layout with numerical values marked in green rectangles

2.2 Nonfunctional requirements

While functional requirements specify what a software system must do, nonfunctional requirements dictate how it should perform those functions. These requirements govern the qualities and attributes that define the system's overall behavior, performance, and user experience.

N1. Performance

The module should process the video stream at a rate of at least 2-3 times faster than real-time playback. This ensures that the Raspberry Pi 4 device can handle the processing load without experiencing delays or interruptions in the stream.

N2. Hardware Compatibility

In the hospital, the module will operate on Raspberry Pi 4 device. Therefore, ensure compatibility with Raspberry Pi 4 hardware, optimizing performance for this specific platform. The module should leverage the capabilities of the Raspberry Pi 4 to achieve optimal efficiency and resource utilization.

N3. Configurability

Module parameters and settings should be configurable via a dedicated config folder. This allows for easy customization and adjustment of settings without the need for code modification, enhancing flexibility and ease of maintenance.

N4. Reliability

The module will have to operate continuously for longer periods of time. Therefore, it should be robust and reliable, capable of continuous operation without frequent failures or crashes. It should handle unexpected scenarios gracefully, ensuring uninterrupted processing of the video stream.

2.3 System architecture

The main focus of this work is VentVision, a module for the GStreamer library that parses relevant data from the lung ventilator screen. We have already mentioned all that, but now we will look at how such a module actually works so that we can design our own.

2.3.1 GStreamer module

GStreamer is a C library that supports video (non-linear editing) processing. [18] This processing is executed via pipelines which are constructed from modules. In the Figure 2.2 you can see a simple scheme of a GStreamer pipeline that consist of a datastream source, a filter that processes the data and a sink which receives the processed data and saves or displays them.

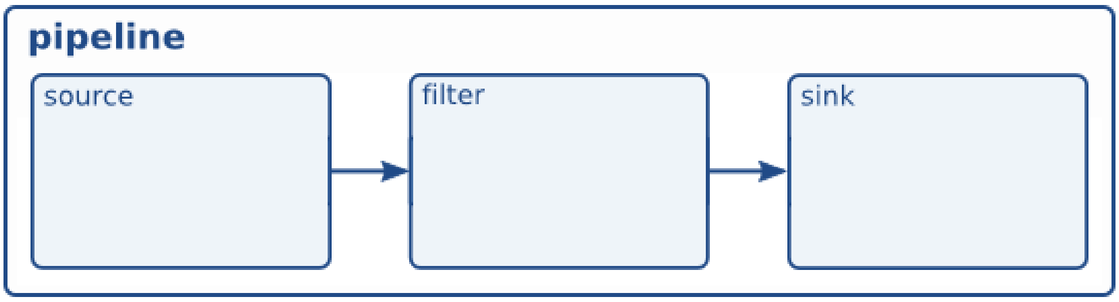


Figure 2.2 Simple GStreamer pipeline obtained from the official Gstreamer website [19]

The VentVision module would be a filter in the simple scheme 2.2. That means, it receives the video stream frames one by one from the previous module, which may be a source or another filter, then automatically calls a function for each frame to process it, and sends the processed frame to the next module, which may be a sink or yet another filter.

The functioning of the GStreamer module is based on three main parts, namely the initialization of the module, then the actual frame processing function and finally the deinitialization of the module. The **initialization** of the module is done when the pipeline starts and is used to load data from the configuration folder, allocate resources and check that everything is set up correctly. The **frame processing function** should then consist of these tasks:

1. Check if the curves should be parsed and exported.

   Curves should be exported if they are fully loaded in the current frame. But since they always stay loaded for several frames in a row before they reset and start loading again from the beginning, we only need to export the first of those frames to avoid sending out redundant data. Therefore, as can be seen in the Figure 2.3, the curves will only return to the export-ready state again when the algorithm detects a gap in the curve. Then the next fully loaded curve is exported again.

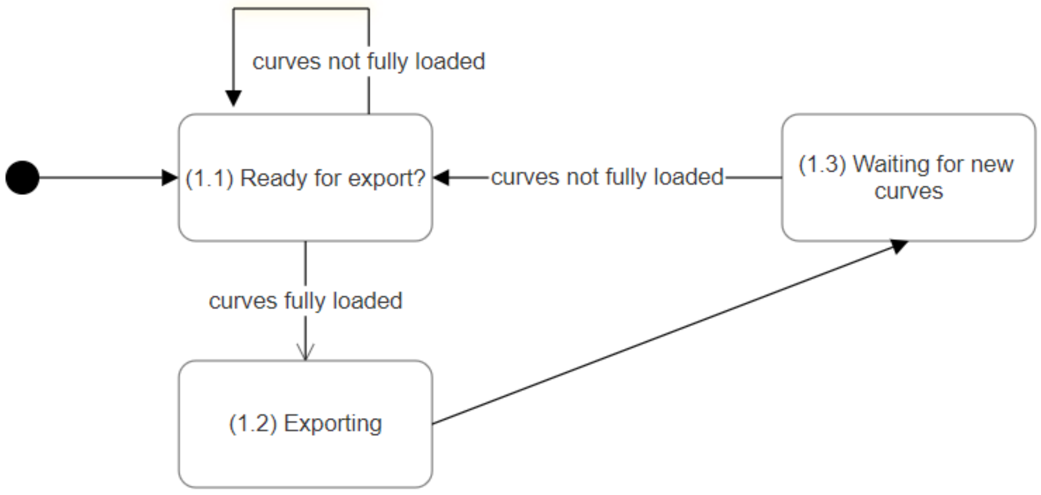


Figure 2.3 State machine diagram, that shows, when the curves should be exported

2. Parse the data from the frame.

   If the data should be exported according to the first point, then it must be parsed first. Parsing is not necessary if the data from the current frame is not to be exported. The data to be parsed include the Paw (airway pressure) and Flow (*Průtok* in the Figure 2.1) curves, the name of the current mode (which can be seen in the upper right corner in the Figure 2.1 – in this example it is *SIMV*, meaning Synchronized Intermittent Mandatory Ventilation), and the values highlighted in the Figure 2.1.

3. Export the data via signal.

   In order to export the data that we parsed from the frame, we first need to create a single string from the data so that we can send it to the server using a signal. This string should, according to the non-functional requirements, be in csv format, i.e. the values on the line are separated by commas.

Finally, when the video stream ends and the pipeline stops, we must **deinitialize** the module, i.e. release the allocated resources.

2.4 Integration

Now that we know how the VentVision module itself should work, we will look at how it should work in the context of the already existing VENT-CONNECT system.

2.4.1 GStreamer pipeline

We have already discussed what the GStreamer library pipeline looks like and what role the VentVision module plays in it. However, VentVision as a filter (see the Figure 2.2) does not receive the data input directly from the source. In a more complex GStreamer pipeline, there can be multiple filter-like modules that follow after one another. In our case, the VentVision module follows after a filter named GstAnonymizerModule (also referred to as AnonFilter), which makes sure that no personal data are being streamed further.

The GstAnonymizerModule is essential for the VENT-CONNECT system because it takes care of personal data protection. It is a filter that compares each frame with stored templates to recognize by the frame layout which ventilator is connected to the stream. Then, based on that, it covers the places where patient personal data is present in that layout.

The VentVision module works only for the Hamilton lung ventilator, so it is necessary to find out what kind of ventilator is connected to the stream. Fortunately, there is no need to compare the frame with the templates again, because the individual modules in the pipeline can communicate with each other by sending and receiving signals. Thus, once the AnonFilter module has figured out the name of the current layout, it uses a signal to send it to the VentVision module. In the Figure 2.4 you can see how the whole GStreamer pipeline works.

2.4.2 VENT-CONNECT

In the Figure 2.5 you can see the pipeline for the entire VENT-CONNECT system. Based on what we have already said about the VentVision module, it is clear that in this picture it takes care of the condition recognition phase.

The VentVision module will run as part of the GStreamer pipeline on a Raspberry Pi 4 in the hospital. The source of the video stream for the pipeline will be an HDMI input to which the lung ventilator will be connected. After VentVision converts the video data to a time series, it will send it out using a signal that is captured by the server. The server saves the data from the signal as a csv file and then evaluates it using a classifier. For clarity, you can see this process in the sequence diagram in the Figure 2.6.

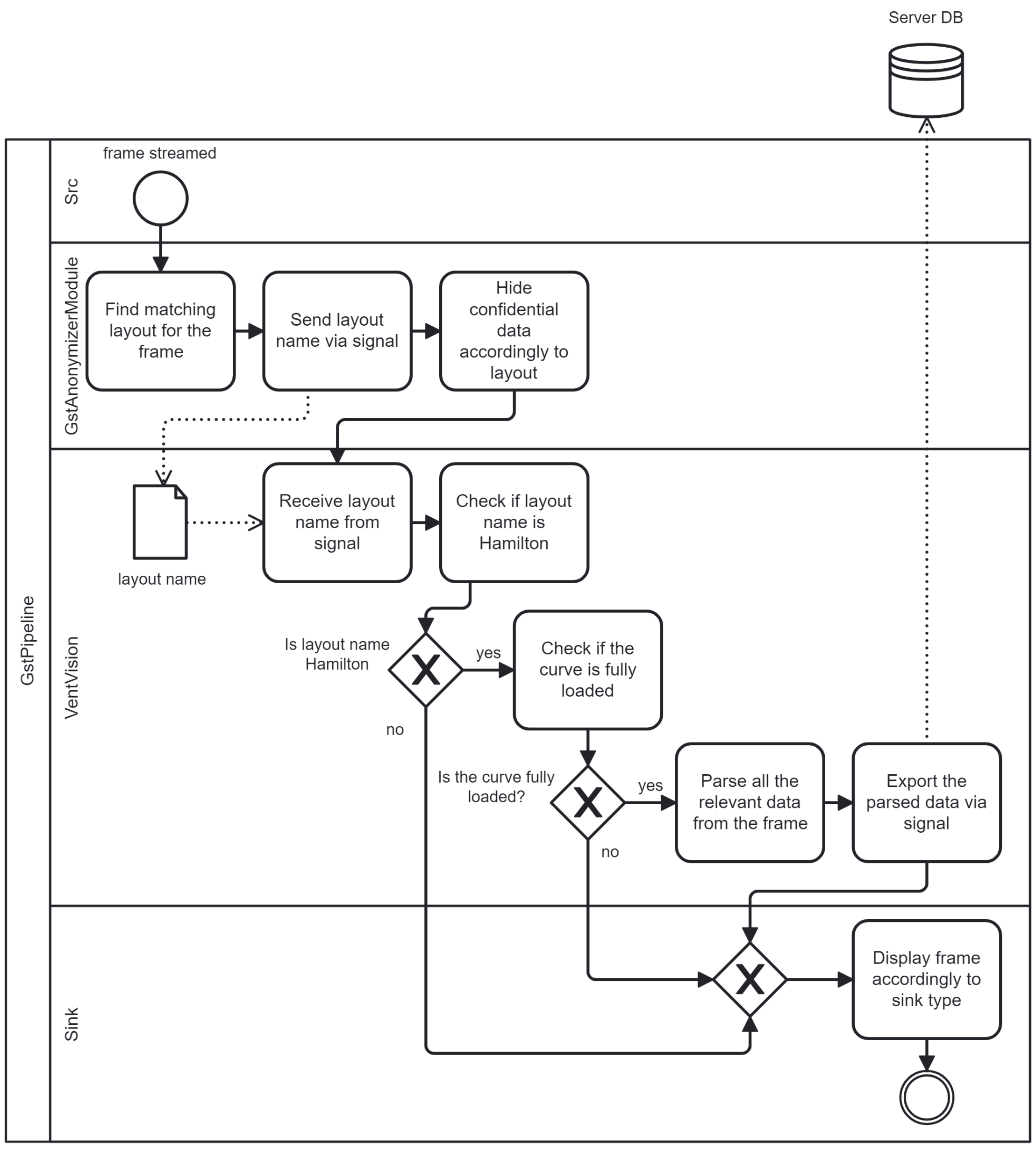

Server DB
GstPipeline
Src
frame streamed
GstAnonymizerModule
Find matching layout for the frame
Send layout name via signal
Hide confidential data accordingly to layout
VentVision
layout name
Receive layout name from signal
Check if layout name is Hamilton
Is layout name Hamilton
yes
no
Check if the curve is fully loaded
Is the curve fully loaded?
yes
no
Parse all the relevant data from the frame
Export the parsed data via signal
Sink
Display frame accordingly to sink type

Figure 2.4 BPMN diagram showing how the project GStreamer pipeline works

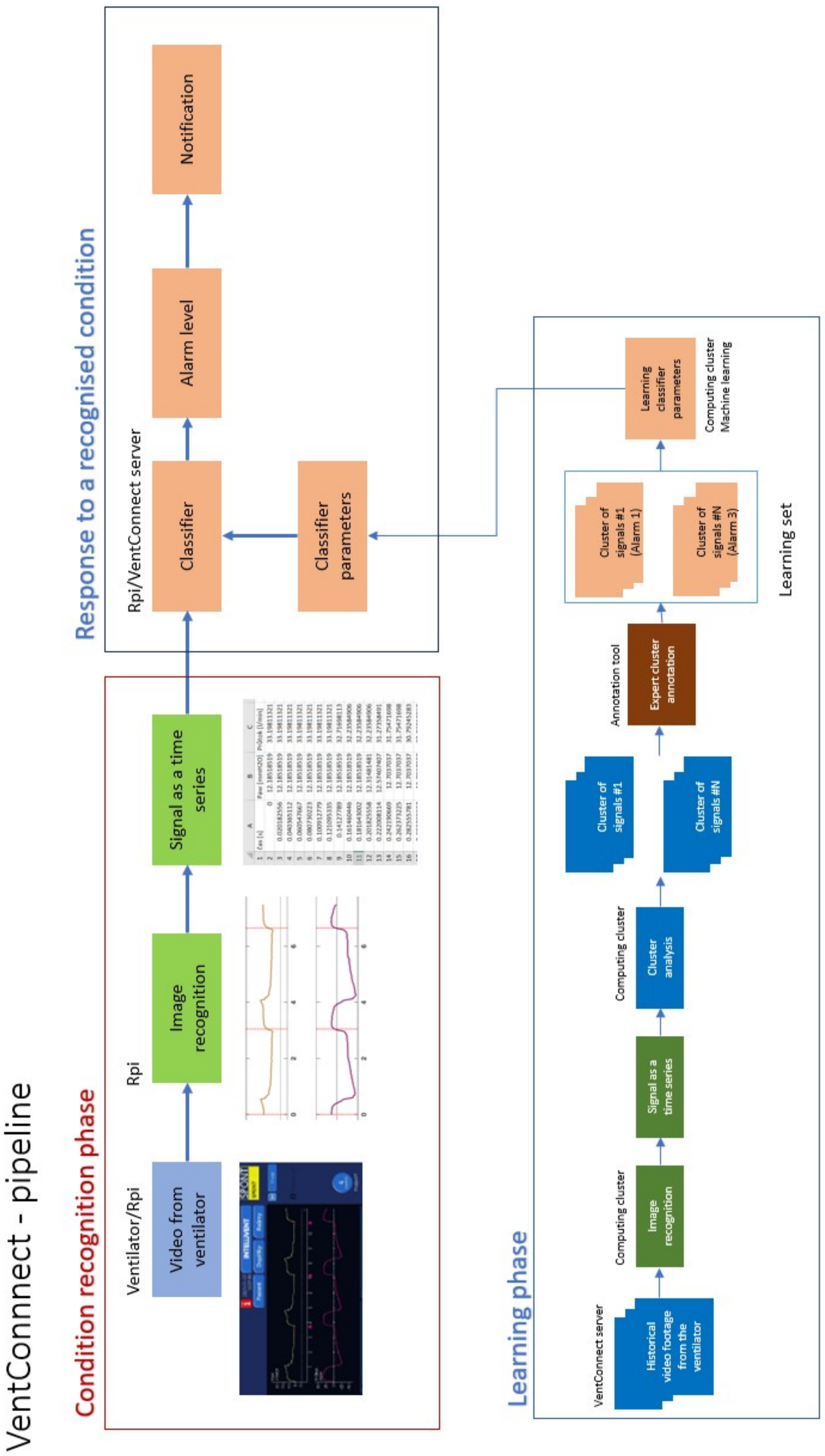


Figure 2.5 Scheme of VENT-CONNECT pipeline by Ing. Miroslav Macík, Ph.D. and Ing. Lenka Vysloužilová, Ph.D.

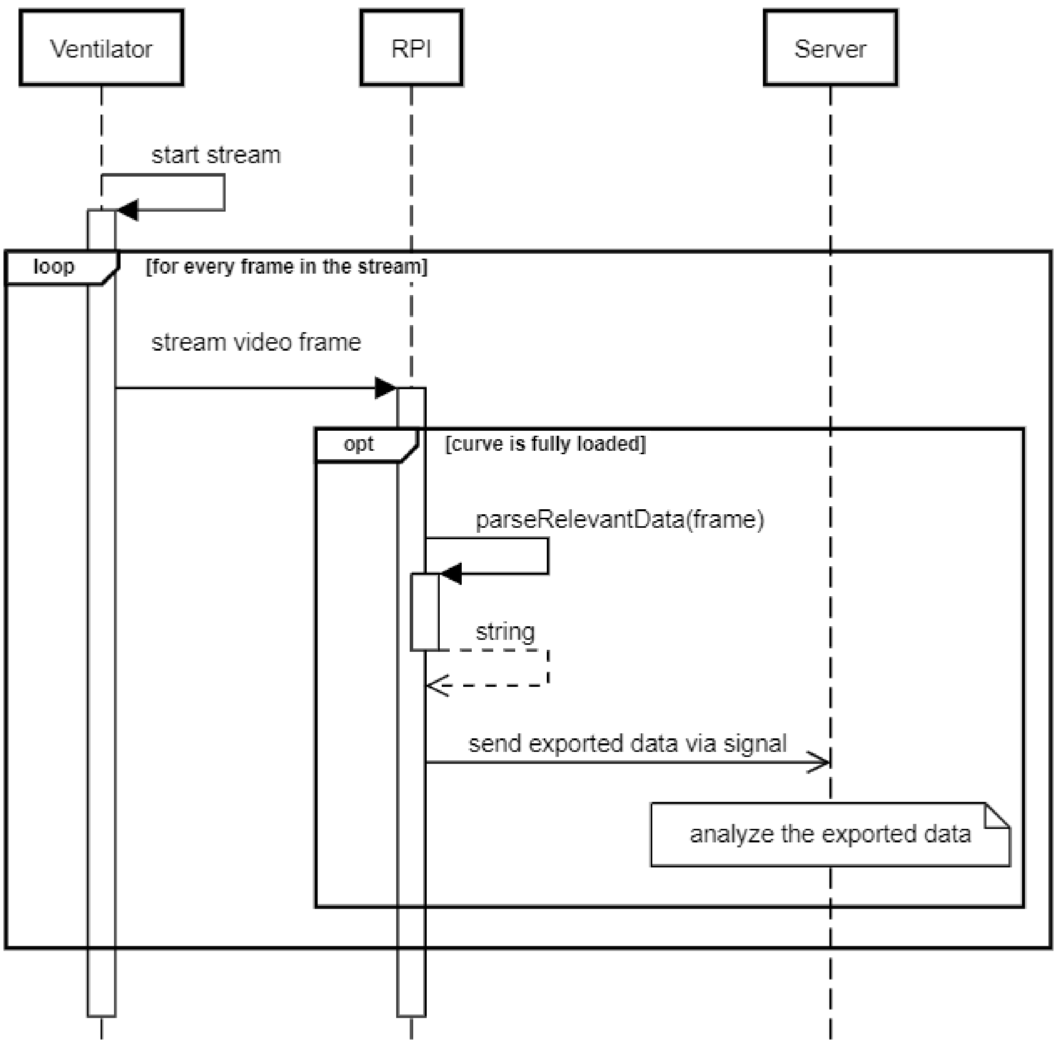


Figure 2.6 Sequence diagram mapping the integration

Chapter 3

Implementation

The implementation can be divided into two parts for simplicity: using the GStreamer library tools and implementing our own algorithms. Although these parts are very much intertwined, we will try to stick to this structure to maintain readability.

3.1 Building GStreamer plugin

How to create the basis of a GStreamer module, otherwise known as a plugin, is described in the instructions in the GStreamer library documentation [20]. We had already a working GStreamer plugin in the project called AnonFilter, which you can read about in the Design chapter. This allowed us to use the skeleton of this plugin to build our own.

The minimum requirements for the very basic functioning of such a plugin are contained in three files: meson.build, gstplugin.c and gstplugin.h. These files are essential for every plugin. The first file mentioned, meson.build, is used to build the project so that it can be compiled, and the folder created by it is then used for the compilation itself. The file contains information about dependencies against the GStreamer library, version requirements, additional libraries if they need to be added, and finally information about the plugin we are creating. We have this information split into two meson.build files – one located in the main project directory, the other in the gst-plugin subdirectory. The first one includes general dependencies, compiler information, etc. In the second one you will find specific information about our plugin, such as its name, current version, required dependencies and versions, etc. Both are necessary to build the project.

As for the latter two files, their names are parameterized and GStreamer expects them based on the information provided by meson.build. We have already passed the name of our plugin to it. Our plugin is called VentVision, so GStreamer expects its source code to be in the gstventvision.c and gstventvision.h files. The header file contains mainly structures, the most important of which for the actual running of the plugin is `_GstVentVision`, which contains an instance of `GstVideoFilter`, since our plugin is specifically a video filter within the GStreamer pipeline. In addition, there is also a structure that contains all the plugin-relevant data that the plugin gets from the configuration folder, and also a structure that stores the data ready for export and more secondary structures.

The C source file is full of functions without which the plugin could not work at all, but which we can leave more or less as a black box and mention them only marginally, because understanding their inner workings is not necessary for the development of our plugin and it is enough to mention their presence. First, there are two functions that are used to construct our filter. The `gst_ventvision_class_init` function, which is used to initialise the class only once (specifying what signals, arguments and virtual functions the class has and more), and the `ventvision_init` function, which is used to initialise a specific instance of this type [20]. The first of these two functions, which initializes the class, does a number of things that need to be listed:

- It takes care of managing properties. That is, it first assigns functions (so called Setter and Getter) to the class that the class will automatically use when it receives an argument to assign it to the correct variable, and later, when it needs to get to its value, it will be able to access it automatically. It also calls a function from the GLib library that allows the class to receive a particular argument. It also assigns a function to the class that releases properties. For these purposes, it treats the class as an instance of GObjectClass from the GLib library.

  We add two properties for our plugin, namely `config-dir`, which contains the path to the folder where the configuration for our plugin is located (this configuration will be described later in our work), and `export-dir`, which contains the path to the folder where the data will be exported as .csv files if the user chooses to use this argument.

- It adds pads to our module. These pads are how the plugin communicates in the pipeline with the surrounding plugins. We need two pads: a source pad, through which our plugin receives the video stream, and a sink pad, through which we send individual frames of the stream to the next plugin. To create the right pads for our plugin, we need to specify the so-called caps. These are the video stream parameters like size, format, colorimetry and other values for which our module is designed. That is also included in this C source file.

- It assigns a function to the plugin that will be called automatically whenever the pipeline state changes. This means, for example, that the pipeline goes into a state where it is ready to play a stream, or that it starts playing, stops playing, etc. This function calls, among other things, the reading of the configuration file and therefore the allocation of resources that are needed to store data from the configuration folder. This happens when the pipeline state transitions from *NULL* to *READY*. On the inverse transition, the allocated resources are released again.

- Creates a new signal that allows the class to emit. This `exported-data` signal is later used to export the parsed data. Its contents are an error flag and a single long string containing all the data to be exported in the same format in which it would be written to a .csv file.

- It connects to the signal sent by AnonFilter so we can see what layout is currently running on the ventilator screen. To do this, we first need to get a pointer to the AnonFilter that is in the same GStreamer pipeline as our plugin, which can easily be done using a library function once we know the name of the pipeline element we are looking for. Then we connect (again by it's name) to the emitted signal and assign a callback function to it, meaning that as soon as AnonFilter sends new information, this function will be automatically triggered in our plugin. We do all this in the `set_info` function, which we assign to our filter here. Again, this is a function that is run automatically by the GStreamer library at the right time.

- Assigns a function to `transform_frame_ip`. This is the function that is responsible for the primary operation of the filter, i.e. it transforms each individual frame in place. The assignment in our module consists of two parts. First, the filter is assigned the `set_info` function, which looks at the filter as our `GstVentVision` object defined in the header file and automatically assigns the process function to the class, which contains our algorithms for solving our problem. Next, it assigns the transform function itself, which is then automatically called by our filter for each frame that passes through our plugin. The function then, upon being called, locks the VentVision object using the GStreamer library's mutex to prevent unwanted serialization during frame processing, calls the `process` function, and unlocks the mutex again. If the AnonFilter signal indicates that we are not monitoring the correct ventilator layout, the processing function will not start.

The `ventvision_init` function does nothing more than call the GStreamer library function, which creates an instance of the plugin and returns information about whether the instance was successfully created.

In this subsection we have discussed the basic functions without which the plugin could not work as part of the Gstreamer library. This information will then serve as the foundation on which the rest of our code is built. Now it is time to take a look at how the data from the configuration folder is processed.

3.2 Processing configuration

We need to get quite a bit from the configuration folder into our code to avoid hard-coded values. The information loaded includes any coordinates in the image, as well as .png files containing templates for recognizing values in the image, and more. In order to more easily describe how the data from the configuration folder is stored and accessed in the code, we will first focus on the tangle of structures we use in our code.

3.2.1 Used data structures

We use a lot of different structures in our work and it might be difficult to find your way around them, so we will now briefly describe what each structure is used for, from the simplest to the more complex, and describe the relationships between them.

Interval: Since most of the data we work with consists of different numerical ranges, the complete basis for us will be a kind of pair replacement that we call an `Interval` and register its start and end.

CropCoords: If we are dealing with a bounded portion of a frame that we are, for example, comparing to a template to recognize text or a value in the frame, we need two intervals, one for the x-axis and one for the y-axis. We wrap these two intervals in a `CropCoords` structure.

Curve: In this structure we have all the information needed to check or parse the curve. We need the coordinates of the part of the image where the curve is located, and then the brightness range within which the pixels of the curve can be found. We will discuss how we compare pixel color later, for now we just need to know that we are only storing information about pixel brightness.

By repeatedly testing the module on different screen captures, we found that sometimes a curve may be overlapped by another curve that may or may not be present on the same graph. In an even worse case, a situation can arise where a curve is overlapped by another curve in one place and intersects the x-axis in another place when it goes negative, and this axis also overlaps it. Because of these findings, we also had to include the brightness range for the secondary curve and the brightness range for the x-axis in the structure.

TemplatePNG: Structure that stores the template read from the .png file. A simple array of color values is enough to contain the template itself. We also need to store the value that is displayed on the template if the template represents a number. Furthermore, we need to know the height and width of the template for further processing. And in some cases, we also need to know the name of the template so that we can assign the correct pointer to it.

TemplateGroup: In most cases it is necessary to work with templates in groups – for example, x-axis labels, y-axis labels, mode names, etc. These groups have common properties that we need to store for them, which include the number of templates in the group, a threshold indicating what percentage of the template must match the image to confirm a match, a background brightness value, a boolean value indicating whether to match the background, and a match tolerance, which is a value for how much the brightness of the image can differ from the brightness of the template to still be considered a match. In addition to all this, `TemplateGroup` of course also holds the `TemplatePNG` array.

Axis: The axis scale calculation is performed independently of the curve parsing function in its own function. In particular, because all curves share a single described x-axis. Thus, it is clear that even for parsing and calculating the scale of each axis, it is necessary to keep its own structure. In this we find the coordinates on which the axis itself is located (better said, the bar just next to the axis on which the ticks can be easily traced), a group of templates for labels, a separately stored template for label *0* which is needed when determining the scale of the y-axis, which can go into negative values. In addition, `Axis` also contains an integer variable into which the position of label *0* is written when it is found.

Last but not least, the `Axis` structure stores a way to calculate the coordinates of the area where we find the label we will compare the template to. This way is different for the x-axis and the y-axis, because depending on the axis, one dimension is always given fixed in the configuration file and one is given relative to the coordinates of the tick it is located at. Therefore, the structure holds fixed the x coordinates of the y-axis labels and the y coordinates of the x-axis labels. In addition, we then find a function in the structure that returns the complete coordinates of the label based on the position of the tick and these fixed coordinates.

LSD: The first of the two most complex structures in the module, LSD stands for Left Side Digits, denoting a group of numeric values on the left side of the screen that need to be digitized because they are relevant to the downstream classifier. You can see these values as the five lines on the left side of the Figure 2.1.

Once again, this is a structure containing all the data one needs to know in order to machine read these values from the screen. This data includes a group of templates for the digits, a group of templates for the text labels that can be used to determine what numeric data we just read, and values that ensure the parsing is done correctly. These are the spacing (number of pixels) between digits, the line spacing, the x coordinate of the last digit on the first line, the y coordinate of the first line, the expected number of lines, the maximum length of the numeric value (calculated per number of characters), the y coordinate of the text label on the first line, and the x coordinate of text labels in general.

RSD: RSD unsurprisingly stands for Right Side Digits. Again, this is a group of values that can be found in the blue circles found on the right side of the ventilator screen (see the Figure 2.1). This structure again contains everything that is needed to read them. The process of parsing them is different from LSD, so a custom structure is required and different values are stored. Again we find a group of templates for digits, a group of templates for text labels and this time additionally a group of templates for units. Then again, the expected number of lines and for a change the brightness range for the presumably white color and the common fairly large coordinates for the area we will be looking for the RSD.

Mode: `Mode` is a fairly simple structure that contains the data needed to recognize the name of the mode in which the ventilator is currently operating. In this context, we need the coordinates where the mode name is located in the frame, a group of templates for each mode, and we also store the mode name where the resulting detected mode is stored.

3.2.2 Configuration folder

Our goal is to get the data contained in the configuration folder into the described data structures in the code. Let us now take a look at what the contents of the configuration folder look like before we discuss how we are going to handle it.

The path to this folder must be passed to the plugin as the `config-dir` argument, otherwise the plugin will end up with an error because it lacks configuration. This folder must contain the config.cfg configuration file, the README.md file that describes the structure of the configuration file, and the templates folder that contains the templates as .png files.

The configuration file is written in the form of commands. On each line to be parsed by the plugin, there is one command that starts with the name of the data structure that will store the data that follows it as arguments to that command. To make it easier to understand, here is a short example of a command combination that might appear in a configuration file, and what its meaning would be:

```
GROUP, x_labels, templates/x_labels, 99, FALSE, 24, 7
  TEMPLATE, x_1,  x_1.png,  1
  TEMPLATE, x_2,  x_2.png,  2
ENDGROUP
```

This part of the configuration file consists of four commands. The first statement named `GROUP` means that we want to achieve the `TemplateGroup` structure in the code. Its first two arguments are the name of the group and the relative path from the configuration folder to the folder where the templates, that this group will contain, are located. The other arguments are the data we already know from the earlier description of data structures: the threshold in percent, the flag if the background is to be compared, the background brightness value, and the brightness value tolerance. The next two commands are `TEMPLATE`, which means that we are creating objects of type `TemplatePNG` that will also be included in the group defined by the first command. Their arguments are, in order: template name, file name, template value (in case it represents a digit, otherwise it is not filled). `TEMPLATE` statement always has to be between `GROUP` and `ENDGROUP` statements. The last `ENDGROUP` statement with no arguments just terminates the group defined by the first statement.

For the sake of completeness, we add here examples of the three remaining commands that appear in the configuration file:

```
VALUE, lsd_space_between_digits, 2
INTERVAL, rsd_white_y_ratio, 227, 237
COORDS, mode, 890, 1020, 5, 30
```

These commands are quite simple and not difficult to navigate. The `VALUE` command only passes an integer value and its name. The `INTERVAL` statement does the same thing, but instead of passing one value, it passes two – a start and an end. And third, `COORDS`, which corresponds to the `CropCoords` structure, does the same thing again, but it arranges two intervals one after the other.

The whole configuration file is just a sequence of these six types of commands, so it is not difficult to add new values to it if needed, just add a new command at the end. Even if we added a command to the configuration file containing data that our plugin does not use at all, nothing would happen, the program would just dynamically allocate additional memory and save the data. We just have to be careful that any `TEMPLATE` statements we enter are in the middle of `GROUP` and `ENDGROUP` statements, and that there is always a file containing the template. We will cover the whole configuration parsing process now.

3.2.3 Configuration parsing

We talked about the data structures we use in our code and we talked about the data provided by the configuration file that we would like to store in these structures. It remains for us to explain how configuration parsing occurs.

In the previous section on the configuration folder and, in particular, the form of the configuration file, you will notice that each entry entered within a configuration command includes its name as the first argument to the command. At the same time, the data structures we have described do not store this name. Therefore, when parsing these commands, we use a super-structure of these structures, usually named `Named<Structure>`, where `<Structure>` is replaced by the name of that particular structure.

Before we get into describing exactly how we parse the configuration, we have two data structures left to mention that have not been relevant so far, but now play a key role.

Configuration: The `Configuration` structure holds all data taken from the configuration folder in dynamically allocated GPtrArray arrays. These arrays are quite simple to handle, as they can be assigned a destructor, i.e. their own free function. However, we do not use them further to actually handle the data in the code, since it is not clear at a glance what the pointers point to without having read all the code in the project. The `Configuration` structure holds four such fields: `named_template_groups`, `named_coords`, `named_intervals` and `named_values`.

It is important to note at this point that any other structures that work with templates, which are large arrays of integer values, only hold pointers to those templates, and the arrays are stored only in the `Configuration` structure.

ConfigurationParsing: This structure holds a pointer to `Configuration`, the path to the configuration folder, a pointer to the momentarily open template group, the line number of the currently parsed line, and finally a simple structure containing the command we are currently processing, split into its arguments. The number of the currently processed line is important only for possible error messages. The rest of the elements will be explained in the following description of the parsing process, if relevant.

Parsing process

The skeleton of the parsing process was again taken from AnonFilter, modified to fit our plugin. The `read_config` function, which takes care of parsing the configuration, is called when the state of the GStreamer pipeline changes from *NULL* to *READY*, which we mentioned in the last section, where we discussed more about the plugin's specifics.

First, `Configuration` and `ConfigurationParsing` are created and destructors are assigned to the dynamically allocated pointer arrays. Next, a configuration file is opened – the name of the configuration file is defined directly in the code and the path to the configuration folder is specified using the `config-dir` property. The lines of the configuration file are processed sequentially. The line is always split first into so-called tokens, which are separated by commas. These tokens are stored as a command in `ConfigurationParsing`. The first token is stored as a command type and based on its value, a function is called to parse the command type.

Each of the functions that parse single command type first checks the corresponding number of arguments to the command type, then checks to see if there is an open template group. An open template group must exist for the `TEMPLATE` and `ENDGROUP` commands, but must not exist for the remaining commands. If the arguments can be matched to the expected variable types in the structure to be created, then a new named structure is created and added to the corresponding pointer array in `Configuration`, and in the case of `GROUP` command, a pointer to the group is also added as an open group to `ConfigurationParsing`. This is only the case for `GROUP`, `COORDS`, `INTERVAL` and `VALUE`, while `TemplatePNG` structure created by `TEMPLATE` is added to the open template group in `ConfigurationParsing` and `ENDGROUP` does not even create any structure at all, so it just sets the open group to `NULL` and returns.

Part of the `TEMPLATE` command processing is of course also loading the .png file into the integer array. For this we use the libpng library [21]. We check that the image is in grayscale format. When we created these templates, we only used the brightness level of the individual pixels as their grayscale value. Therefore, we expect the image to be in this format again and we only want one value from each pixel.

After all the lines of the configuration file are processed, the only thing left to do is to check if any groups have been left open. This completes the parsing of the configuration. Then you need to split the data stored in `Configuration` based on its names into the correct structures that are part of the plugin.

3.3 Frame processing

Finally, we come to the issue of frame processing itself, which means analysing and parsing all the relevant data from the ventilator screen. This involves the following steps, which we will discuss in detail:

1. Check if it is possible to parse the curve.
2. Parse the curve.
3. Parse side digits (numerical values from the screen).
4. Export the data.

Now we describe how we check if the curve is fully loaded.

3.3.1 Checking curve

So far, we still have not properly described how we look at individual pixels in an image. We certainly have to do it now. Frames are pipelined by GStreamer to our plugin as pointers to a `GstVideoFrame` objects, which are actually images in UYVY color format. This format uses 8 bits per pixel location to encode the Y channel (also called the luma channel), and uses 8 bits per sample to encode each U or V chroma sample. However, it uses only 16 bits per pixel instead of 24, because it contains fewer samples of U and V than of Y. [22] This can be seen in the Figure 3.1.

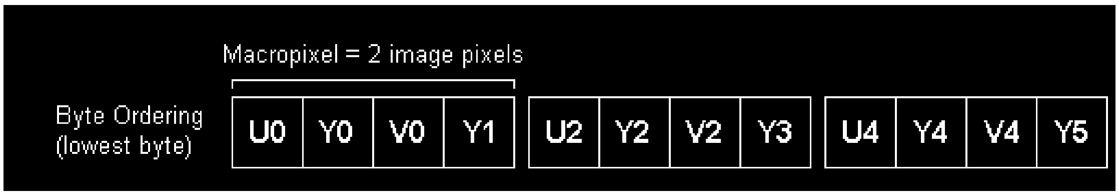


Figure 3.1 UYVY color format illustration acquired from website [23]

During the development of the plugin, we observed the values of the individual channels of the UYVY frame format and came to the conclusion that the Y channel of the pixels is sufficient to distinguish the individual pixel colors. This was beneficial to us because the Y channel of the entire image can be easily extracted by the GStreamer library function into a simple array of pixels and then we just browse through it normally. The brightness level given by the Y channel has been verified through the whole project as sufficient information for us to use it to distinguish different colours of neighbouring pixels. However, this is only true for uncompressed footage. Fortunately, the primary function of our plugin is to run in real-time on the live feed from the fan, where no compression occurs.

Now that we know what we are using to determine pixel color, let us move on to a description of curve checking. This is done in the way we illustrated in the Figure 2.3. Our plugin includes a state machine for each curve that is checked – these machines are always made up of a single boolean variable that tells us whether we want to parse the next fully loaded curve. We extract the Y channel from the image, and use set of for loops to loop through the coordinates bounded by the curve's `CropCoords` that we got from the configuration file. We traverse the cropped area of the image column by column (columns of pixels) from left to right, looking for the running cursor (see the Figure 1.2).

The running cursor is recognized by a column that does not contain a single curve pixel. We have talked about the ways in which a curve can be overlapped with something else. Therefore, we need to check that the column contains neither a pixel of the curve nor anything else by which the curve is overlapped. If we find the running cursor, the curve is not fully loaded. This means that we will not parse it from the current frame, but it also means that the next frame with a fully loaded curve will be the first fully loaded one since the last export, which is why we want to export it. So once we find the running cursor, we set the state machine to ready to export.

Sometimes it happened that the search for the running cursor was disturbed by the legend in the upper left corner (see the Figure 1.2), so we decided not to check the whole curve, but only the second half of it, from which we get the same results.

If the running cursor is not found, it means that the curve can be considered fully loaded. Since we are only looking at the second half of the curve, it may happen that on all frames where the running cursor is on the first half of the curve, the curve will be marked as fully loaded. Because of this, however, we have our state machine set to not export the next fully loaded curve once we find the first fully loaded curve in the series.

If the state machine is ready to parse the curve and the curve is fully loaded in the same frame, continue to parse the curve and eventually export the data. This process is done separately for each single curve.

3.3.2 Parsing curve

Parsing a curve is several levels more complex than just checking it. This can be broken down into several steps, the first few taking care of the preparation so that parsing can be done, and then the actual parsing. Let us write it again in the form of numbered points for clarity:

1. **Removing legend:**

   Now we need to get rid of a legend that might cause noise in our data without losing the data from the associated part of the graph. So the very first step to successful curve parsing is to delete the legend. We need to check, if the legend is really there, because it is not always. We have loaded the text template, coordinates and all other configuration information, and now we will compare it to the location in the image where it should be. Since the curve can pass directly through the legend, we will use the option to ignore the background color and only look for a match of the text itself when comparing. To evaluate the match, we use the dice coefficient calculation mentioned above and the threshold from the configuration that the dice coefficient must overcome to be considered a match.

   Let us say we found a match and now we want to remove the legend. To do this, we use a simple function that goes through the cropped portion of the image again according to the template and assigns a background color to each pixel in the image that does not have a background color according to the template.

2. **Calculating axis scale:**

   You will notice on the Hamilton ventilators that there is only one common set of x-axis labels for both curves. Therefore, the x-axis scale for both curves is calculated from the same Axis structure. The y-axis exists for each curve separately but at the same time on the same x-coordinate and their positions differ only in the y-coordinate, which does not need to be specified in the configuration and is derived from the region where the curve is observed.

   In the case of the x-axis, proceed through the cropped area from left to right, in the case of the y-axis from bottom to top, to first discover label *0* and its position before calculating the scale. If we know the template for label *0*, we compare it from the beginning to each label we find until we find a match. If we do not find a match, we assume that zero is at the origin of the graph. We append each tick we find on the axis to a linked list.

   Once we reach the end of the axis, we start comparing the label at the last tick with the label templates. Some labels differ very little from each other, for example only by a sign, so we compare the location in the frame with all the templates in the group and then ask whether the highest match achieved was sufficient, i.e. higher than the required threshold. If it is not sufficient, this could be due to two different reasons: we do not have a template stored for this label, or the label is overlapped by something. Both of these problems are solved in most cases by going back to the previous label and trying to match on it. We repeat this process until we find a sufficient match, or until we run out of ticks.

When we find a sufficient match, we take the distance between the tick where the match was found and zero (the origin or stored position for zero) and divide that distance by the numeric value of the template where the match was detected. In doing so, we assume that the distances between ticks are equal and the scale does not change and is uniform across the axis width. We denote the obtained proportion as the calculated scale of the axis.

3. **Parsing curve:**

To get to this point, both axes must have their scales calculated correctly. When parsing the curve, we go through the coordinates of the cropped area in the same way as when checking the curve. This time, however, there are a few things that can go wrong and that we need to be careful to resolve.

It is possible that there is more than one pixel of the curve in each x-coordinate (especially where the curve rises or falls rapidly), but we are only storing one value for one x-coordinate in our time series, so we need to average the y-coordinate. If something overlaps the curve, we store a pixel of that instead of a pixel of the curve on that x-coordinate. However, it may happen that at the same time the curve is overlapped by the x-axis, we are finding pixels of the secondary curve and have to decide which one should fill in for the curve. In such a case, we choose the position that is closer to the y coordinate that we calculated at the previous position, since we do not know for sure which one overlapped the curve. When calculating the y-position of a pixel, we also need to subtract the stored position of the zero on the y-axis, since we can move in negative values.

There is one more possible problem we have encountered. This occurs when the curve passes through the legend. The legend must be removed, otherwise it interferes with the averaging of the y position of the curve. However, while removing it, a hole may be accidentally created in the curve if the curve has passed directly through the legend. For these cases, we have implemented an algorithm that fills these gaps. Once we discover a hole in the curve, we simply calculate the size of the hole and use the surrounding values to fill it in, increasing or decreasing linearly as needed.

Finally, we divide each x-position by the x-axis scale and each y-position by the y-axis scale and store these values in the time series. After we parse the curve, we set a flag to let us know that it is possible to export data from this curve.

3.3.3 Parsing side digits

Parsing of the side digits happens when flags for both curves are set to ready for export. Before we can export the data, we still need to parse the relevant numerical data displayed on the sides of the ventilator screen (see the Figure 2.1). We already got acquainted with LSD and RSD when describing data structures. Even then it was obvious that different data is needed to parse them, and therefore it cannot be done in the same way for both. We will first describe parsing LSD.

LSD (Left Side Digits)

On the ventilator screen in the Figure 2.1, you may notice that the LSD are right-aligned while their labels are left-aligned. Individual digits (or even other mathematical characters) have regular spaces between them when their templates are correctly framed. Of course, we could extend the templates by a few more pixels so that we do not have to worry about spacing at all, but then we would have to compare larger templates. So we use both regular spacing between digits on a single line and regular spacing between lines to move from digit to digit. Equally large line spacing allows us to move between text labels as well.

We will not not prolong the discussion of the parsing itself unnecessarily, since it is done in the expected way by searching for template matches. We will, however, mention some special features of LSD parsing. For example, the numeric values we parse are not just integers, nor just floats, but may contain, for example, a colon, since it may be a ratio. Therefore, it is not possible to parse them into a simple float variable, but you must use an array of chars.

Another thing is that numeric values on the screen can change color. For example, one line can be red or yellow at a time and its Y channel suddenly does not match the templates. Instead of adding a bunch of unnecessary templates of digits in different colors, in this case we decided to replace the function that compares the image to the template with a function that only compares the shape to the template. That is, it verifies that what is supposed to be the background is the background, and what is not supposed to be the background is not the background, but it no longer deals with the colors that make up what is not the background.

The values we get from parsing are stored in pairs with the name of the associated label.

RSD (Right Side Digits)

At first glance, the RSD may appear to be aligned to the center of the area where they are located, and the line spacing may appear regular, just like the LSD. However, the opposite is true. During the creation of the templates, we found that the RSD do not have regular spacing between them, but can be offset by one or more pixels. Therefore, we proceeded to a different algorithm for parsing them.

From the configuration, we obtain the coordinates of the entire screen area where the RSD may occur and traverse it from below. We look for a white pixel and once we find one, we run the `get_label_height` function on that line to find out how tall the text to which the white pixel belongs is. This function has the nifty side effect of finding the x-coordinate where the text starts (its leftmost x-coordinate). Based on the height of the text and this x-coordinate, we can now determine the coordinates at which to compare the image to the templates.

From the bottom to the top, the three types of text we find along the way are continually sorted in this order: the type of value being monitored, the units, and the value itself. We therefore use a state machine using an enum to parse. To make sure we are not just observing some noise, we must always find a match to a template, otherwise we do not move to the next state, but stay in the same one.

By the way, we also use the `get_label_height` function between each digit of a numeric value, just because it finds the first pixel of the text. When we correctly recognize the numerical value, we save it together with the type and unit as a triple among the data intended for export.

3.3.4 Exporting data

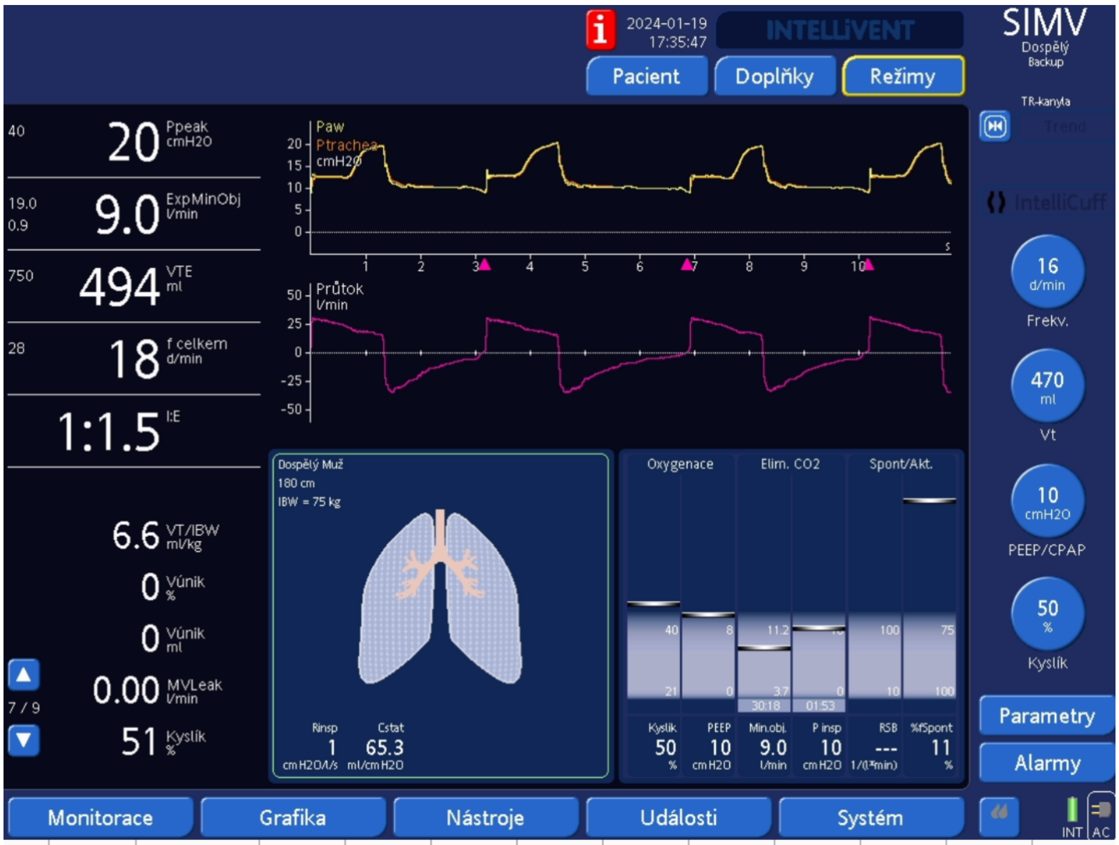
Let us take a look at what data we are exporting. In this order they are: timestamp, mode, Paw curve time series, Flow curve time series, LSD and RSD. When the ventilator screen displays a warning, we write that to the output as well, but that is not relevant to our work at the moment. You may notice that we have not mentioned mode parsing and timestamp calculation up to this point. Both are performed directly before the actual export and are trivial operations.

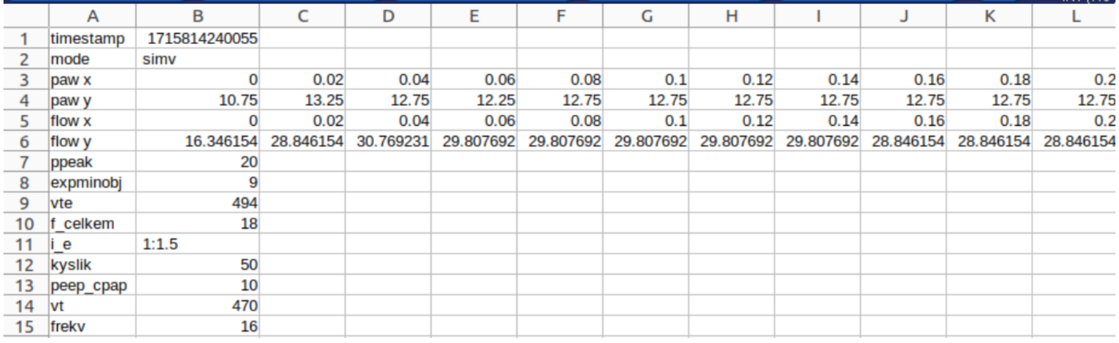
All the data to be exported is sequentially added to a single long string in the order we specified above. In this string each special item is on its own line (lines are separated by '`\n`' of course), the first column is the header, and the next columns are the associated data. The columns are separated by commas as this is the same format used by .csv files.

All that remains to be done is to send a signal containing the output string, and if the `export-dir` property is set, also create a .csv file to write this string to. Then it it just a matter of freeing the memory and preparing the structures we used to export so they can be reused.

We must also mention that the .csv file format assumes by default that the header of the file is the first row and not the first column. It would be more complicated for us to list the data in such a way, and it is not necessary. When necessary, this data can be transposed using a Python [24] script that we also created and is included in the utils folder in the attached repository. However, this will not be done on the VENT-UNIT device.

To give you a better idea of what the output of our described module looks like, in Figure 5, you can see a snapshot of the video input alongside its associated output, generated directly from this video frame.





	A	B	C	D	E	F	G	H	I	J	K	L
1	timestamp	1715814240055										
2	mode	simv										
3	paw x	0	0.02	0.04	0.06	0.08	0.1	0.12	0.14	0.16	0.18	0.2
4	paw y	10.75	13.25	12.75	12.25	12.75	12.75	12.75	12.75	12.75	12.75	12.75
5	flow x	0	0.02	0.04	0.06	0.08	0.1	0.12	0.14	0.16	0.18	0.2
6	flow y	16.346154	28.846154	30.769231	29.807692	29.807692	29.807692	29.807692	29.807692	28.846154	28.846154	28.846154
7	ppeak	20										
8	expminobj	9										
9	vte	494										
10	f_celkem	18										
11	i_e	1:1.5										
12	kyslik	50										
13	peep_cpap	10										
14	vt	470										
15	frekv	16										

Figure 3.2 Example of input and its assosiated output.

Chapter 4 Evaluation

The content of this chapter is a critical assessment of the effectiveness and performance of the module developed in this thesis. The main objective is to present a comprehensive comparison of the newly developed module with a previously implemented algorithm addressing the same problem domain. In addition, this chapter defines the integration tests performed to ensure the smooth functioning of the module as a whole. It also discusses the crucial phase of testing in a clinical setting, specifically in a hospital setting where the module is expected to be deployed.

4.1 Comparison to the earlier solution

In our analysis, we recognize the prior work of Ing. Milan Nemy, Ph.D., which we detailed in the Analysis chapter. His MATLAB-based solution, like ours, involved parsing curves from the Hamilton ventilator screen into time series. This made it possible to compare our module's results with his by computing the Pearson correlation coefficient (PCC) [25] for each exported curve. This comparison is essential for evaluating the effectiveness and improvements of our module. Since our solution runs in real-time on the RPIs in the hospital, while this earlier solution has to be manually run by the developer, we will henceforth refer to our solution as the on-line algorithm and the earlier solution as the off-line algorithm for the purposes of this comparison.

The Pearson correlation coefficient (PCC) serves as a statistical tool for assessing the likelihood of a connection or relationship between two zero-mean real-valued random variables. Ranging from -1 to 1, approaching 0 indicates independence between the variables, while approaching 1 indicates a stronger correlation. Essentialy, the closer the value of PCC is to 1, the more are the variables likely to be dependent on each other, showing a stronger relationship between them. [25]

It can be calculated using the following formula:

$$\rho(a,b) = \frac{E(a,b)}{\sigma_a \sigma_b}$$

provided by Cohen [25]. Which can be simplified to the following formula for even better readability:

$$\rho(a,b) = \frac{\sum_{i=1}^{N}(a_i - \bar{a})(b_i - \bar{b})}{\sqrt{\sum_{i=1}^{N}(a_i - \bar{a})^2 \sum_{i=1}^{N}(b_i - \bar{b})^2}}$$

where N is the number of values of the variables, which both have to contain the same number of values otherwise the formula is not applicable, $\bar{a}$ is the a average and $\bar{b}$ is the b average.

Another relevant aspect for our evaluation is the p-value. Basically, the p-value offers a way to estimate how well a particular dataset aligns with a proposed model. Typically, this model includes certain assumptions along with what is known as a ”null hypothesis.” Often, this null hypothesis suggests the absence of an effect, such as no difference between two groups or no relationship between a factor and an outcome. The smaller the p-value, the stronger the evidence against the null hypothesis. Essentially, a small p-value raises doubts about or provides evidence against the null hypothesis and its underlying assumptions. In statistical analysis, a commonly used threshold for statistical significance is a p-value less than 0.05. [26]

In our case, our null hypothesis assumes that the Pearson correlation coefficient (PCC) is equal to 0, implying no correlation between the variables. By determining the p-value, we aim to demonstrate that it is highly improbable for the observed correlation in our results to occur purely by chance. This helps strengthen our argument that the correlation is indeed meaningful and not just a random occurrence.

We conducted an evaluation by visually inspecting the on-line algorithm’s output from 30 video record samples. Subsequently, these outputs were compared with the results obtained from processing the same samples using the off-line algorithm. It is worth noting that in five instances, the off-line algorithm encountered difficulties in parsing the curves from the video for various reasons. For the remaining samples, we calculated both the Pearson correlation coefficient (PCC) and the corresponding p-value. This statistical analysis allowed us to quantitatively assess the degree of correlation between the outputs of the online and offline algorithms.

Sample number	**Modality**	**PCC**	**p-value**
⋮	⋮	⋮	⋮
4	paw	0.9998	<1e-16
	flow	0.9998	<1e-16
5	paw	0.9999	<1e-16
	flow	0.9998	<1e-16
6	paw	0.9895	<1e-16
	flow	0.9998	<1e-16
⋮	⋮	⋮	⋮
27	paw	0.9889	<1e-16
	flow	0.9996	<1e-16
⋮	⋮	⋮	⋮
Mean		0.996556	
Std		0.007374975891	

Table 4.1 Section of analysis of on-line and off-line algorithm correlation.

In the Table 4.1, you can observe a section of table detailing the calculated PCC and corresponding p-values from all 30 samples (25 excluding the ones the offline algorithm could not solve). Below the table, the final mean and standard deviation of all PCC values are computed.

Given the notably high average PCC value, the results suggest a strong correlation between the outputs. The p-values, all $< 1e - 16$, are significantly smaller than 0.05, indicating high statistical significance.

In the Figure 4.1, you can examine time series plots of the off-line and on-line algorithms. Sample 27, chosen specifically for this purpose due to its comparatively lower correlation, illustrates one of the improvements offered by the on-line algorithm. As depicted in the initial portion of the first graph, the legend is incorrectly removed in the off-line plot, causing the curve values to fluctuate, which should not occur.

Among other observed improvements provided by the on-line algorithm are the ability to parse Paw curves with negative values, the ability to parse curves overlapped by the x-axis, numerical values detection and more.

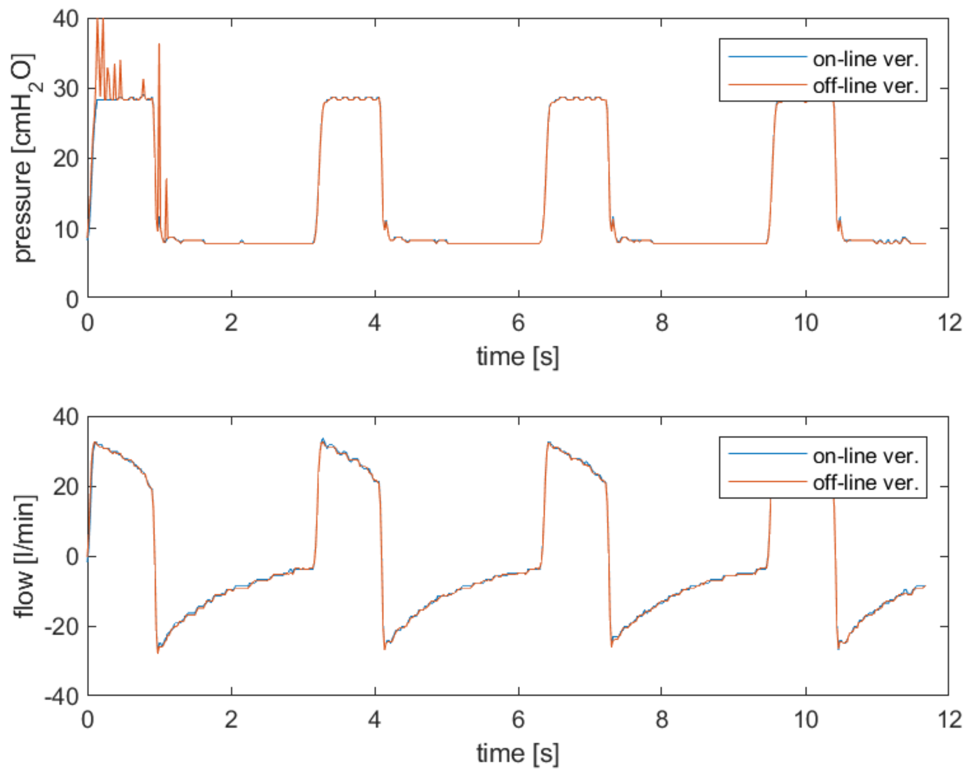


Figure 4.1 Plots of on-line and off-line outputs.

The results of correlating with the off-line algorithm suggest that our solution performs equally well, if not better, than the existing solution for the same problem within the whole project by VENT-CONNECT team. You can find the complete table in the appendix, from which we have presented a portion alongside associated graphs for each sample.

4.2 Integration tests

To verify the functionality of our solution and ensure that no errors were introduced into the code after each modification, we created a set of integration tests. These tests use as input data nearly all samples whose outputs were “verified” in the previous section of this chapter. We consider these outputs as correct, and after code modification, we can check if the algorithm returns them identically.

We utilized the Python testing framework Pytest [27] to create a template using parameters for test creation. Thus, the number of tests changes depending on how many correctly named samples (according to the expected structure of the input data) you add. Currently, we are testing with input video recordings in .mkv format, which we store in compressed form to save space (using lossless compression), decompress within the testing script, and compress within the testing script again after testing. The testing script also allows the use of .mp4 files as input data, but we cannot guarantee the success of their testing as they are compressed video recordings that our algorithm may not work for. In the appendix, you can find the test.py file in the repository's test folder, which contains these Pytest tests.

Additionally, this script tests another required aspect of our solution – speed. When you run this script, it generates, among other things, a .csv file containing processing time data for individual samples. When executed on a regular computer, this data merely reveals to developers whether the algorithm has significantly slowed down since the last testing. However, our interest lies in the processing speed on the RPI, where we discovered that processing one sample takes our algorithm 3-4 seconds, while one sample lasts for 20 seconds. This processing speed more than meets the requirements.

4.3 Testing plugin in hospital

The final and most crucial part of this chapter is the testing conducted directly in a hospital environment. On Monday, April 22, 2024, we, together with the VENT-CONNECT team, conducted the first test of the VentVision module in the hospital. Instead of testing directly on a patient, we used an artificial lung. From this testing, we obtained a set of output data from our module in the form of .csv files, as well as data from the DATALOGGER [28] software, which we consider as ground truth for testing our module.

When dividing the number of pixels by the scale, our module obtained a new value every 20 ms, while DATALOGGER only provided data every 32 ms. As a result, we needed to interpolate the output of our module to align it with the frequency of the DATALOGGER data for correlation calculations. In the Figure 5, you can visually compare the outputs generated by VentVision with those from DATALOGGER.

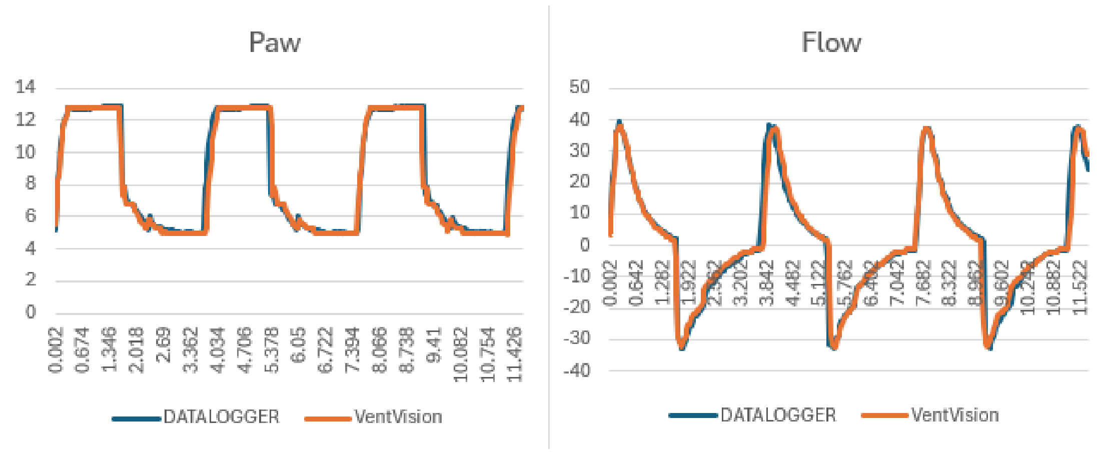


Figure 4.2 Plots of VentVision and DATALOGGER outputs.

In the appendix, you will find the remaining data from the mentioned testing conducted in the hospital. As evident from the comparison of graphs in the Figure 4.2, the testing of the module yielded quite successful results. It can be positively stated that the scales correspond to the ground truth from DATALOGGER. The Pearson correlation coefficient (PCC) for both curves was around 0.98, and the p-value is again significantly smaller than 0.05 by several decimal places. Therefore, it can be concluded that the correlation also turned out in our favor.

Following this successful testing, the VentVision module was deployed on almost all devices VENT-UNIT in the hospital and is currently operational on live patients in the testing phase.

We randomly selected some recordings from the VENT-CONNECT system, which you will find in the appendix, and verified the accuracy of our module's results with them. This method of testing also serves as a reliability test for the module, ensuring that it can continuously operate for longer time periods without interruption.

Chapter 5

Conclusion

In our work, we focused on developing the VentVision module, which aims to digitize various relevant data from the screens of Hamilton ventilators, particularly curve profiles. Let us revisit the specific objectives that framed our work and evaluate whether we successfully achieved them. These are the goals that we outlined in the introduction of our work:

1. *Examine the current implementation and development of the VENT-CONNECT system, focusing on modules related to the VentVision component. Investigate the AnonFilter module, which directly precedes VentVision, and analyze their interconnections. Describe the functionalities of the GStreamer library.*

   We examined the current state of the VENT-CONNECT system in the Analysis chapter, where we provided detailed descriptions of its components, and in the Design chapter, where we discussed how the VentVision module should integrate into this system. We focused on the AnonFilter module primarily in the Design chapter, outlining its connection with the VentVision module. Finally, we described the GStreamer library and its functionalities in various sections across the Design and Implementation chapters.

2. *Identify optimal ways to leverage GStreamer for analyzing image data from lung ventilator screens. Focus on extraction and analysis of curves and other relevant data (alphanumerical values, alarms, etc.).*

   To fulfill this task, we utilized the opportunity to draw inspiration from the AnonFilter and its use of the GStreamer library, as well as tutorials available in the GStreamer documentation, which we referenced in our work. Although the GStreamer library does not directly provide functions to fulfill the main task of our work (which was parsing relevant data), its capabilities such as signal transmission and reception made our work much easier.

3. *Develop the VentVision module emphasizing parsing and analyzing data obtained from the ventilator screen. Ensure seamless integration with the existing AnonFilter module and efficient utilization of the GStreamer library.*

   This task is the focus of entire chapters Design and Implementation. The integration with the existing AnonFilter module through signals is enabled and tested. The Evaluation chapter, where we assess the results of our work, demonstrates that this task has been successfully accomplished as expected.

4. *Evaluate the VentVision module capabilities of exporting analyzed data to facilitate its utilization by other components within the VENT-CONNECT system.*

   The Evaluation chapter focused precisely on this topic. It demonstrated through available means that the accuracy of our solution is highly probable. Additionally, it showed that our solution is sufficiently fast and capable of operating continuously for entire days as part of the VENT-CONNECT system.

5. *Integrate the finalized VentVision module into the VENT-CONNECT system, ensuring compatibility with the AnonFilter module and establishing a reliable mechanism for exporting parsed and analyzed data for further system utilization.*

   The VentVision module has been successfully integrated into the VENT-CONNECT system, as mentioned in the previous point. At this moment, our module is running in a testing phase on devices with live patients, digitizing data from ventilators at their bedsides. It sends output data via a signal to the server, where they are stored for comparison with screen recordings. The data export mechanism has also been tested.

As we reach the conclusion of our work, our primary aim has been to ease the burden on overstretched hospital staff and thereby improve healthcare, not only in our country but beyond. Currently, we are still in the testing phase of our module, but we believe it is on the right track to become a valuable asset in the future. We plan to further develop our module to make it adaptable for various types of devices in intensive care units. We look forward to a time when our efforts will make daily tasks easier for doctors and hopefully ease tensions in hospitals, especially in times like the COVID-19 pandemic.

Bibliography

1. *Number of COVID-19 patients in hospital — ourworldindata.org* [https://ourworldindata.org/grapher/current-covid-patients-hospital]. [N.d.]. [Accessed 21-04-2024].
2. MÖHLENKAMP, Stefan; THIELE, Holger. Ventilation of COVID-19 patients in intensive care units. *Herz*. 2020, vol. 45, no. 4, pp. 329–331.
3. VANEK, Jakub; MACIK, Miroslav. VentConnect, system for remote monitoring of medical instruments. In: *Proceedings of the 24th Bilateral Student Workshop CTU Prague and HTW Dresden-User Interfaces & Visualization, Dresden, Germany*. 2018, pp. 12–16.
4. VYSLOUZILOVA, Lenka; ZVONICEK, Vaclav; DUSKA, Frantisek; JIRMAN, Jan; KUBR, Jan; LHOTSKA, Lenka; MACIK, Miroslav; NEMY, Milan; NIEBAUEROVÁ, Eliška; POVIŠER, Lukáš; SAMEK, Martin; VANĚK, Jakub. NOVÁ TECHNOLOGIE V INTENZIVNÍ PÉČI POMÁHÁ VZDÁLENĚ SLEDOVAT PACIENTY NEJEN S COVID-19. *Medsoft*. 2021, vol. 33, pp. 84–87. Available from DOI: 10.35191/medsoft_2021_1_33_84_87.
5. *HAMILTON-G5/S1 - The modular high-end ventilation solution — Hamilton Medical — hamilton-medical.com* [https://www.hamilton-medical.com/en/Products/HAMILTON-G5-S1.html]. [N.d.]. [Accessed 24-04-2024].
6. HAMILTON MEDICAL AG. *HAMILTON G5: Operator's Manual*. 2012. 624074/07 Software version 2.2X.
7. JIRMAN, Jan; MACIK, Miroslav. VentConnect: live to life and the octopus in the hospital server room. In: *Proceedings of the 25th Bilateral Student Workshop CTU Prague and HTW Dresden-User Interfaces & Visualization, Dresden, Germany*. 2022, pp. 60–65.
8. MATLAB. *version 9.0.0 (R2024a)*. Natick, Massachusetts: The MathWorks Inc., 2024.
9. FOUNDATION, Raspberry Pi. *Raspberry Pi 4 Model B* [https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf]. 2019.
10. MALKADI, Abdulkarim; ALAHMADI, Mohammad; HAIDUC, Sonia. A study on the accuracy of ocr engines for source code transcription from programming screencasts. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 65–75.
11. *GOCR — jocr.sourceforge.net* [https://jocr.sourceforge.net]. [N.d.]. [Accessed 05-05-2024].
12. DHIMAN, Shivani; SINGH, A. Tesseract vs gocr a comparative study. *International Journal of Recent Technology and Engineering*. 2013, vol. 2, no. 4, p. 80.
13. *GitHub - tesseract-ocr/tesseract: Tesseract Open Source OCR Engine (main repository) — github.com* [https://github.com/tesseract-ocr/tesseract]. [N.d.]. [Accessed 06-05-2024].

14. SMITH, Ray. An overview of the Tesseract OCR engine. In: *Ninth international conference on document analysis and recognition (ICDAR 2007)*. IEEE, 2007, vol. 2, pp. 629–633.

15. KATO, Hajime; NAKAZAWA, Mitsuru; YANG, Hsuan-Kung; CHEN, Mark; STENGER, Björn. Parsing line chart images using linear programming. In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2022, pp. 2109–2118.

16. KUMAR, Anukriti; GANU, Tanuja; GUHA, Saikat. ChartParser: Automatic Chart Parsing for Print-Impaired. *arXiv preprint arXiv:2211.08863*. 2022.

17. SHAMIR, Reuben R; DUCHIN, Yuval; KIM, Jinyoung; SAPIRO, Guillermo; HAREL, Noam. Continuous dice coefficient: a method for evaluating probabilistic segmentations. *arXiv preprint arXiv:1906.11031*. 2019.

18. *GStreamer: open source multimedia framework* [`https://gstreamer.freedesktop.org/`]. [N.d.]. (Accessed on 02/28/2024).

19. *Basic tutorial 2: GStreamer concepts* [`https://gstreamer.freedesktop.org/documentation/tutorials/basic/concepts.html?gi-language=c`]. [N.d.]. (Accessed on 02/28/2024).

20. *Constructing the Boilerplate — gstreamer.freedesktop.org* [`https://gstreamer.freedesktop.org/documentation/plugin-development/basics/boiler.html?gi-language=c`]. [N.d.]. [Accessed 08-05-2024].

21. *libpng Home Page — libpng.org* [`http://www.libpng.org/pub/png/libpng.html`]. [N.d.]. [Accessed 12-05-2024].

22. DREWBATGIT. *Recommended 8-Bit YUV Formats for Video Rendering - Win32 apps — learn.microsoft.com* [`https://learn.microsoft.com/en-us/windows/win32/medfound/recommended-8-bit-yuv-formats-for-video-rendering?redirectedfrom=MSDN#uyvy`]. [N.d.]. [Accessed 13-05-2024].

23. *UYVY yuv pixel format — fourcc.org* [`https://fourcc.org/pixel-format/yuv-uyvy/`]. [N.d.]. [Accessed 13-05-2024].

24. VAN ROSSUM, Guido; DRAKE, Fred L. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN 1441412697.

25. COHEN, Israel; HUANG, Yiteng; CHEN, Jingdong; BENESTY, Jacob; BENESTY, Jacob; CHEN, Jingdong; HUANG, Yiteng; COHEN, Israel. Pearson correlation coefficient. *Noise reduction in speech processing*. 2009, pp. 1–4.

26. WASSERSTEIN, Ronald L; LAZAR, Nicole A. *The ASA statement on p-values: context, process, and purpose*. Vol. 70. Taylor & Francis, 2016. No. 2.

27. *Full pytest documentation &#x2014; pytest documentation — docs.pytest.org* [`https://docs.pytest.org/en/latest/contents.html`]. [N.d.]. [Accessed 15-05-2024].

28. *Datalogger user's guide — Hamilton Medical — hamilton-medical.com* [`https://www.hamilton-medical.com/en_US/Resource-center.html?resource-detail-type=document&resource-detail-id=c3746986-aa43-4a82-8fa9-32f8ec8c19a9`]. [N.d.]. [Accessed 15-05-2024].

Contents of the attachment

```
VentVision
├── README.md .................. instructions for compiling and running the plugin and tests
├── meson.build ............................................. file used to build the plugin
├── gst-plugin ........................................... source code files for the plugin
│   ├── meson.build
│   ├── gstventvision.h
│   ├── gstventvision.c
│   └── ...
├── gst-app ........ source code files for a GStreamer application for testing signal emitting
│   ├── meson.build
│   └── src
│       └── easy_test.c
└── test
    ├── test.py ................................ Python test script using Pytest framework
    ├── *.py ................... Python source code for functions used within the test script
    └── export_examples ................................... input directory for test script
text
├── thesis.pdf ........................................ text of the thesis in PDF format
└── thesis.zip ................................. source form of the thesis in LaTeX format
VV_testing_22-04-24.zip ... output from DATALOGGER (DL) and VentVision (VV) and
their comparison
hospital_live_testing
├── VV_output_files.zip .... csv output files from VV live testing currently taking place in
│   hospital
└── screen_recordings .. multiple ventilator screen recordings that can be matched with the
    VV output
```