Bachelor's Thesis

Visualization Verification of Complex Avionic Models Using Computer Vision

Franz Köhler

Period: 07.10.2024 – 07.02.2025

Supervisor: Andreas Waldvogel

Institute of Aircraft Systems
University of Stuttgart
Professor Dr.-Ing. Björn Annighöfer

Sperrvermerk

Diese vorliegende Abschlussarbeit ist eine nicht Ãűffentliche Version und beinhaltet vertrauliche und interne Daten der MUSTERFIRMA. Sie darf nur der PrÃijfungskommission der UniversitÃďt Stuttgart und fÃijr das PrÃijfungsverfahren notwendigen Personen zugÃďnglich gemacht werden. Die Abschrift bzw. VervielfÃďltigung von Datenmaterial - auch in AuszÃijgen - ob in digitaler oder analoger Form ist nicht gestattet. Eine Ausnahme von dieser Regelung bedarf der schriftlichen Genehmigung des Autors und der MUSTERFIRMA.

University of Stuttgart
Germany

Institute of Aircraft Systems

Directors
Prof. Björn Annighöfer
Prof. Zamira Daw

Contact
Andreas Waldvogel
Pfaffenwaldring 27
70569 Stuttgart • Germany
T +49 711 685-62703
F +49 711 685-62771
e-mail:
andreas.waldvogel@ils.uni-stuttgart.de

https://www.ils.uni-stuttgart.de/

Task description

Bachelor Thesis

Visualization Verification of Complex Avionic Models Using Computer Vision

2024-10-07

Thesis Task Description

In safety-critical systems, domain-specific modeling (DSM) still lacks sufficient automation, often requiring extensive manual effort. A key challenge is ensuring that visual representations are accurate without depending on manual verification processes. This raises the critical question: “Does what you see truly reflect what you get?”
The Institute of Aircraft Systems is conducting research into automated verification of visualizations, with a focus on block diagrams. As part of this effort, a proof of concept has been developed using a simple domain-specific language for the Functions Layer of the Open Avionics Architecture Model (OAAM).
The concept aims to verify whether a screenshot accurately represents a user model by relying on a visualization-defining model. The process involves the following steps:
1. Preprocessing: The screenshot is processed to isolate the block diagram and remove user interface artifacts.
2. Tokenization: The system identifies token types as specified in the visualization model.
3. Syntactical Analysis: The relative positioning of tokens is evaluated using the visualization model.
4. Model Instantiation: Based on the visualization model, the recognized tokens are assembled into a reconstructed model.
Finally, the reconstructed model is compared to the original model. Errors in visualization are identified and reported, both in a detailed report and, if feasible, as graphical overlays on the original model.

While the proof of concept demonstrates feasibility, it is limited to the basic Functions Layer and has been tested on only 14 cases.

Objective

The objective of this thesis is to extend and generalize the proof of concept to accommodate more domain-specific languages and additional test cases. In particular, the Hardware Editor and Allocations Editor—both of which feature more complex visualizations and hierarchies—shall be addressed.

Key tasks include exploring the limits of the existing concept, developing test cases that showcase the enhanced capabilities of the implementation and extending the verification approach to cover the broader scope of domain-specific languages.

Additionally, the thesis requires thorough documentation of the work and findings, as well as a final presentation to summarize the results.

Work items:

- Familiarization
  - Domain-specific modeling
  - Python
  - Computer Vision
  - limits of existing concept
- Concept development
  - Recognition of bigger block diagrams in the Functions Editor
  - Recognition of Intersections
  - Recognition of block diagrams with other token types
  - Recognition of rotated text in the Hardware Editor
  - Recognition of complex block diagrams using nested vertices and edges in the Allocations Editor
- Implementation of the solution
  - Implement the developed concepts
  - Where feasible, generalize the concepts for various block diagram languages
- Demonstration
  - Implement test cases showing the capabilities of implemented functionality
  - Validation and evaluation of the test cases
- Discussion
  - Evaluation of the feasibility of the provided proof of concepts and discussion of further improvements
- Documentation of the results
- Final presentation

Begin:	2024-10-07
End:	2025-02-07
Supervisor:	Andreas Waldvogel
Examiner:	Prof. Björn Annighöfer

Legal provisions: In principle, the student is not entitled to publish any work and research results of which he/she becomes aware during thesis to third parties without the permission of the supervisor. Concerning achieved research results the law about copyright and used protective right (“Bundesgesetzblatt” I/S. 1273, “Urheberschutzgesetz” of 09.09.1965) is valid. The student has the right to publish his/her findings, as far as no findings and achievements of the supervising institutes and companies are included. The guidelines for the preparation of the Bachelor's/Master's thesis issued by the field of study as well as the examination regulations must be considered.

SelbstÃďndigkeitserklÃďrung

Hiermit versichere ich, dass ich diese Bachelorarbeit / Masterarbeit selbstÃďndig mit UnterstÃijtzung des Betreuers/ der Betreuer angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit oder wesentliche Bestandteile davon sind weder an dieser noch an einer anderen Bildungseinrichtung bereits zur Erlangung eines Abschlusses eingereicht worden.

Ich erklÃďre weiterhin, bei der Erstellung der Arbeit die einschlÃďgigen Bestimmungen zum Urheberschutz fremder BeitrÃďge entsprechend den Regeln guter wissenschaftlicher Praxis[1] eingehalten zu haben. Soweit meine Arbeit fremde BeitrÃďge (z.B. Bilder, Zeichnungen , Testpassagen etc.) enthÃďlt, habe ich diese BeitrÃďge als solche gekennzeichnet (Zitat, Quellenangaben) und eventuell erforderlich gewordenen Zustimmungen der Urheber zu Nutzung dieser BeitrÃďge in meiner Arbeit eingeholt. Mit ist bekannt, dass ich im Falle einer schuldhaften Verletzung dieser Pflichten die daraus entstehenden Konsequenzen zu tragen habe.

Stuttgart, den 07.10.2024

Franz KÃűhler

[1] Nachzulesen in den DFG-Empfehlungen zur "'Sicherung guter wissenschaftlicher Praxis"' bzw. in der Satzung der UniversitÃďt Stuttgart zur "'Sicherung der IntegritÃďt wissenschaftlicher Praxis und zum Umgang mit Fehlerverhalten in der Wissenschaft"'

NutzungsrechterklÃďrung

Hiermit erklÃďre ich mich damit einverstanden, dass meine Bachelorarbeit / Masterarbeit zum Thema:

Visualization Verification of Complex Avionic Models Using Computer Vision

in der Institutsbibliothek des Institutes fÃijr Luftfahrtsysteme mit sofortiger Wirkung Ãűffentlich zugÃďnglich aufbewahrt und die Arbeit auf der Institutswebseite sowie im Online-Katalog der UniversitÃďtsbibliothek erfasst wird. Letzteres bedeutet eine dauerhafte, weltweite Sichtbarkeit der bibliographischen Daten der Arbeit (Titel, Autor, Erscheinungsjahr, etc.).

Nach Abschluss der Arbeit werde ich zu diesem Zweck meinem Betreuer neben dem PrÃijfexemplar eine weitere gedruckte sowie eine digitale Fassung Ãijbergeben.

Der UniversitÃďt Stuttgart Ãijbertrage ich das Eigentum an diesen zusÃďtzlichen Fassungen und rÃďume dem Institut fÃijr Luftfahrtsysteme an dieser Arbeit und an den im Rahmen dieser Arbeit von mir erzeugten Arbeitsergebnissen ein kostenloses, zeitlich und Ãűrtlich unbeschrÃďnktes, einfaches Nutzungsrecht fÃijr Zwecke der Forschung und der Lehre ein. Falls in Zusammenhang mit der Arbeit Nutzungsrechtsvereinbarungen des Instituts mit Dritten bestehen, gelten diese Vereinbarungen auch fÃijr die im Rahmen dieser Arbeit entstandenen Arbeitsergebnisse.

Stuttgart, den 07.10.2024

Franz KÃűhler

Abstract

Visualization Verification of Complex Avionic Models Using Computer Vision

With the growing complexity of models and applications in aviation, use of domain specific modeling (DSM) in the field has become vital. It enables engineers to work more efficiently and, through automatic code generation, significantly reduces the number of errors in the resulting programs. For use in safety-critical applications however, DSM requires significant effort. One important issue is ensuring correct model visualization to reduce the amount of manual verification work.
This paper aims to build upon **ar__prof__paper** to improve the reliability of the automated verification of block-diagram visualizations in DSM. In **ar__prof__paper**, a computer vision library called "Open Computer Vision" (OpenCV) is used to recognize and process block diagram models. the recognized Data is compared with the original model to find and indicate deviations to the user inside a browser-based graphical model editor called "eXtensible Graphical EMOF Editor" or "XGEE".
This paper extends the capabilities of the block diagram recognition algorithm to work with complex and diverse diagrams in three graphical domain-specific languages within XGEE. The new implementation is able to correctly identify and process intersecting or partially obscured lines in any orientation, detect a larger variety of vertices and process text labels in multiple orientations, showcasing its potential to significantly reduce manual verification effort in DSM applications.

Kurzzusammenfassung

Verifikation von Visualisierungen von komplexen Avionik Modellen mit Computer Vision

Mit der steigenden KomplexitÃďt von Modellen und Applikationen wird die Nutzung von Domain Specific Modeling (DSM) in der Luftfahrtindustrie immer wichtiger. Es ermÃűglicht Ingenieuren, effizienter und sicherer zu arbeiten und reduziert duch autmatische Code Generation die FehleranfÃďlligkeit der fertigen Programme. Bei sicherheitskritischen Anwendungen jedoch ist DSM mit signifikantem Mehraufwand verbunden. Ein wichtiges Problem ist die korrekte visuelle Darstellung des Modells um manuelle verifizierung zu vermeiden.
Diese Arbeit baut auf **ar_prof_paper**, um die Sicherheit der automatischen Verifikation von Block-Diagramm-Visualisierungen in DSM zu verbessern. In **ar_prof_paper** werden methoden aus der Computer-Vision genutzt, um Block Diagramme zu erkennen, mit den internen Modellen zu vergleichen und Unterschiede visuell darzustellen.

Contents

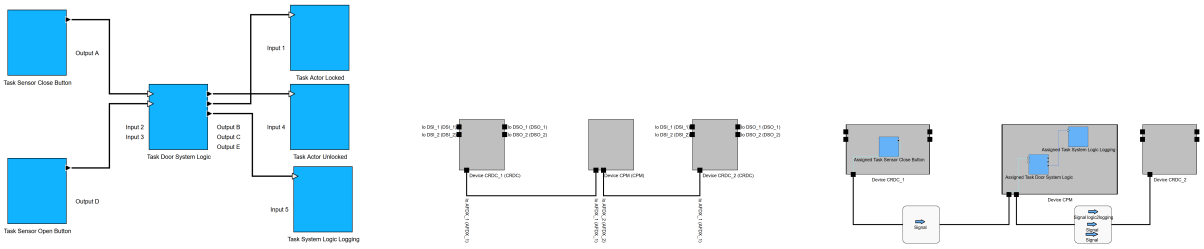
List of Figures

1 Introduction

Development of complex systems requires an efficient way for engineers to ensure the correctness of the developed software. Domain Specific Modeling (DSM) can, through automatic code generation and block-diagram visualizations, significantly reduce the risk of faulty applications, and enable engineers to work more effectively. In safety-critical development processes however, DSM can only be used without subsequent manual verification, if the DSM tools work correctly. This can either be achieved through time intensive qualified software development processes, which ensure accurate and reliable visualization of DSM through the tool itself, or through the use of unverified DSM tools followed by the subsequent use of a small visualization verification tool to ensure the correctness of the application.
In low-cost projects with high safety requirements, a cost-effective qualification method is crucial, highlighting the potential of the latter approach for a qualifiable graphical editing tool for use in DSM.

The Institute for Aircraft Systems is currently developing a way for avionic models to be developed inside a web-based graphical model editor called "eXtensible Graphical EMOF Editor" or "XGEE".
XGEE uses three kinds of "tokens" within three distinct editors to visualize avionic models. They are:

- signals
- vertices
- text labels



(a) Functions editor (b) Hardware editor (c) Allocations editor

Figure 1.1: Functions- Hardware- and Allocations editors

This paper builds upon the work of **ar__prof__paper** to automate the verification process within XGEE by tokenizing a screenshot from within the editor. To recognize and process the screenshot data, methods from the "Open Computer Vision" (OpenCV) library are being used. By rebuilding a model from the recognized tokens and comparing it to the original, visualization errors become apparent and can be indicated to the user. Common visualization errors include:

- unclear signal intersections
- signals being obscured by blocks
- text labels being obscured by signals or blocks

- blocks being scaled down to the point of disappearing
- blocks obscuring other blocks

1.1 State of the Art

The visualization verification tool in **ar__prof__paper** could preprocess an automatic screenshot, tokenize edges, vertices and labels, rebuild a model based on the recognized tokens and compare the recognized model with the original.
To preprocess the screenshot, the XGEE window is found by searching for the distinct colors of the XGEE title- and status bar and defining the bounding box of the actual block diagram. The grey pixels of the grid are replaced with white, since they are not relevant for the model tokenization.

The vertex detection algorithm was able to detect blue function containers, inputs and outputs with great accuracy, as long as they did not exceed the size limit set by the used template. The original edge detection algorithm was able to detect edges that went from left to right and from top to bottom with high accuracy. The original text detection was using PyTesseract and worked very well, if the text was oriented properly.

1.2 Computer Vision

1.3 Domain Specific Modeling

1.4 XGEE

1.5 Python

1.6 OpenCV

2 New Concepts

The XGEE visualization verification algorithms are essential for interpreting and optimizing diagram structures within the editor. While the original implementations demonstrated functionality in basic test cases, their limitations prevented comprehensive detection across various diagram types and complexities. This chapter introduces enhanced methods to address these limitations and significantly increase detection accuracy, versatility and stability.

2.1 Edge Detection

Edges represent connections between vertices, inputs and outputs. Detecting edges accurately requires an approach that can identify individual line segments and how they connect to form complex chains. This section introduces a robust edge detection pipeline leveraging multiple computer vision techniques to reliably identify edges across a wide variety of block diagrams within XGEE.

2.1.1 Methology

Since there are only vertical and horizontal edges within XGEE, two kernels will be used to find all pixels containing either part of a vertical or horizontal edge. The *filter2D()* function places the kernel anchor (usually the top left value of the kernel) on top of a pixel, with the rest of the kernel overlapping the corresponding local pixels.
The kernel values are then multiplied by the corresponding pixel values underneath and added together. The result is saved and placed on the location of the anchor. The kernel on the right 2.1 is rotated by 90°to generate the horizontal kernel. The same process can be written as equation 2.1, where H is the resulting matrix, I is the original image and K is the used kernel. This process is repeated for every pixel and, depending on the structure of the kernel, it can also be used to blur or sharpen an image [1].

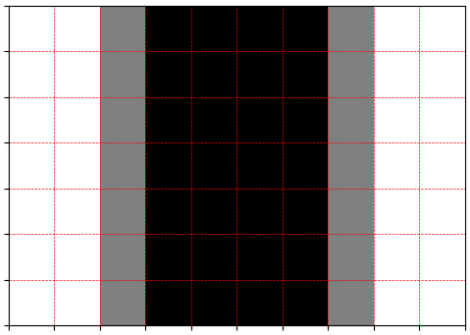


Figure 2.1: kernel for vertical edges

Each value in the processed image is normalized to an 8-bit integer between 0 and 255. This allows the data to be visualized as a gray scale image 2.2 and processed using OpenCV's *thresholding()* function 2.3. As illustrated in figure 2.4, ports and letters are sometimes misidentified as edges. Additionally, intersections of edges and points where horizontal and vertical edges meet are not immediately detected. These challenging areas will be processed individually in a later step in the edge detection pipeline. Apart from these specific cases, the method effectively extracts all pixels corresponding to vertical and horizontal edges, provided their widths match those of the used kernels.

$$H(x,y) = \sum_{i=0}^{M_i-1} \sum_{j=0}^{M_j-1} I(x+i-a_i, y+j-a_j)K(i,j) \tag{2.1}$$

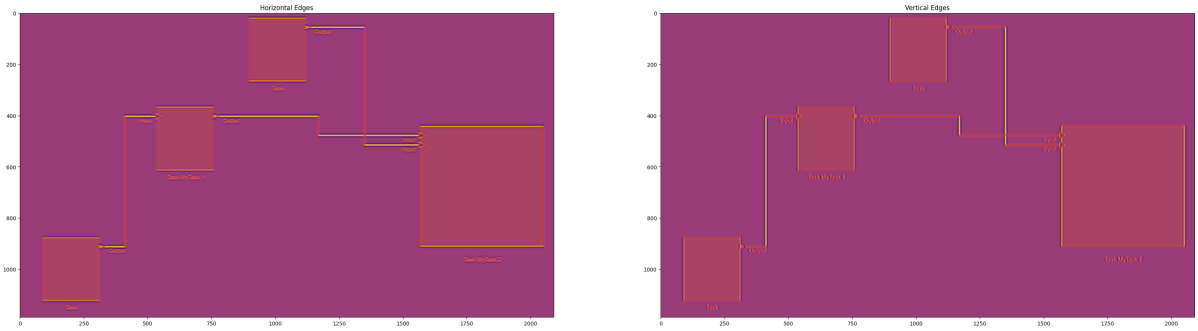


Figure 2.2: Filter2D function results

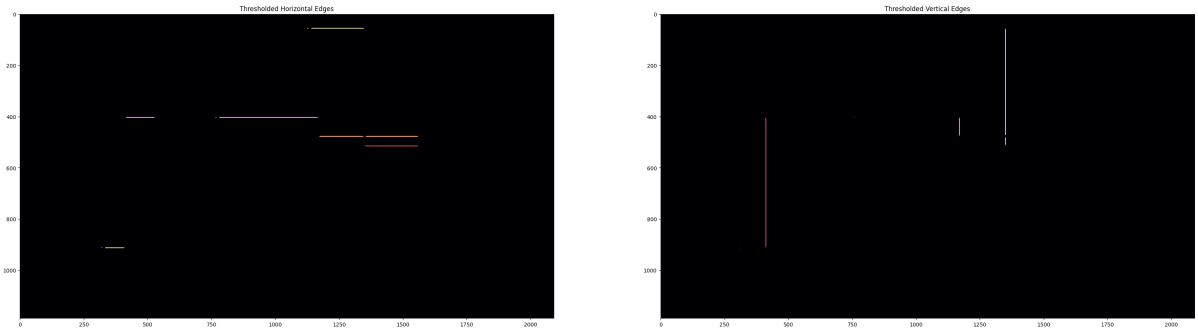


Figure 2.3: Threshholded filter2D function results

For easier processing and data storage, the threshholded pixels 2.3 are converted to line segments consisting of start- and endpoints. This is achieved through two of OpenCV's built-in functions: *findContours()*, which retrieves contours from a binary image using an algorithm introduced in this paper: [2].
approxPolyDP(), which approximates a curve or a polygon with another curve or polygon with less vertices using an algorithm introduced in this paper: [3].

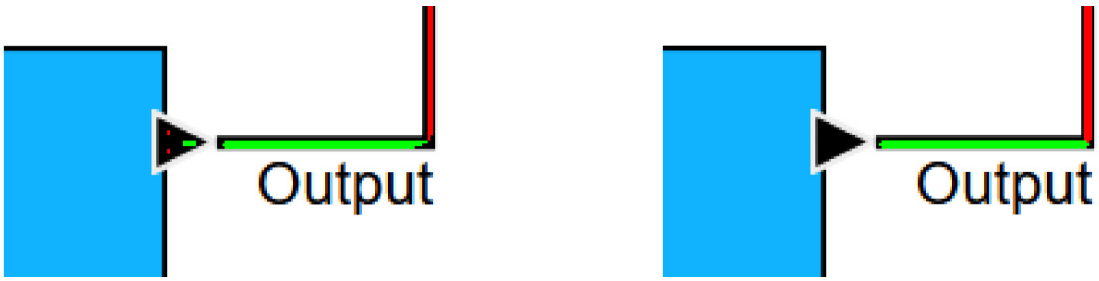


Figure 2.4: Comparison of detected pixels and line segments around an output before and after filtering short segments.

Misidentified letters and ports are removed by filtering out all line segments with a length shorter than 20 pixels. In all test cases, this approach successfully removes the unwanted line segments while keeping the edges 2.4.
As shown in 2.3, the *filter2D()* function initially does not detect any edges at intersections, leading to gaps between the line segments. To process these gaps, it is assumed that intersections always consist of two straight edges. Overlapping 90Âř turns are considered impossible and will result in an error, forcing the user to maintain a clear diagram structure.
First, all intersections in the image are detected using OpenCV's *matchTemplate()* function, which matches a template to overlapping regions of the image (Figure 2.2).
The function slides the template across the image, comparing overlapping patches with the template using a specified method. Among the available methods [4], tm_sqdiff_normed

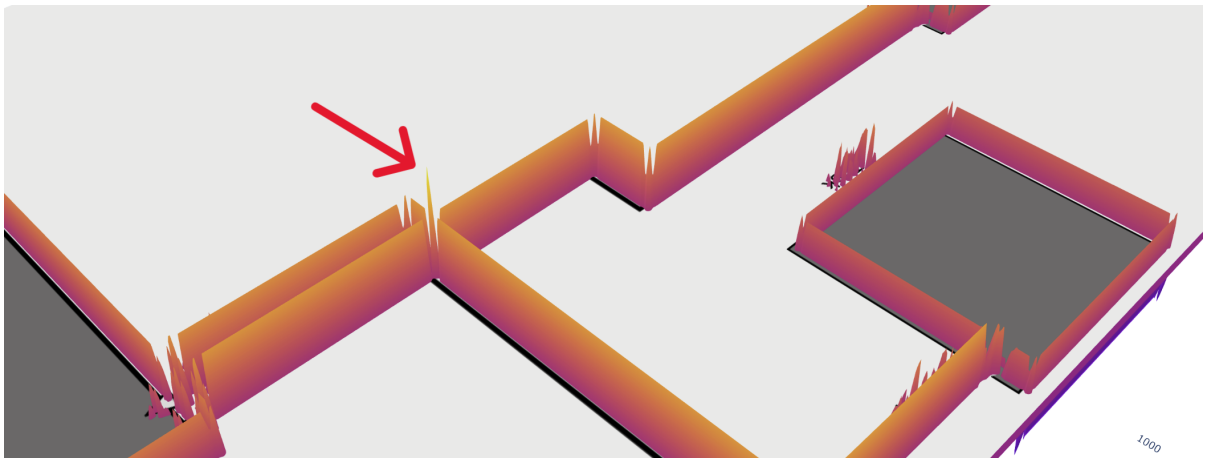


Figure 2.5: Peak is visible at intersection

produced the most accurate results. For each pixel, the function calculates and assigns a value representing the similarity between the template and the corresponding image region. While this approach is more precise than *filter2D()*, it is also significantly more resource-intensive.

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} (T(x',y')^2 * \sum_{x',y'} I(x+x',y+y')^2}} \tag{2.2}$$

At intersections, the similarity value is approximately 85%. This slight discrepancy likely arises from how modern operating systems use aliasing to render text and lines with higher apparent resolution and contrast compared to the display. Zooming in reveals that the white pixels near edges are often replaced with subtle color hues or shades of gray **??**. Rendering the results of the template matching highlights a peak in similarity at the intersection (Figure 2.5). Thresholding isolates this peak, typically yielding two or more matches per intersection. These matches are then filtered based on proximity, ensuring only one match is detected at each intersection.
The method processes each detected intersection by connecting the two vertical and two horizontal line segments. This eliminates any residual points near the intersection, leaving only one vertical and one horizontal line segment 2.7.
In the allocations editor, the same approach is applied to detect and process signal containers on edges. To enhance diagram readability, it is assumed that no 90Âř turns are concealed behind the containers and that edges pass through them in a straight line. The primary distinction from intersection detection lies in the number of line segments: at a signal container, only two line segments meet, rather than four.

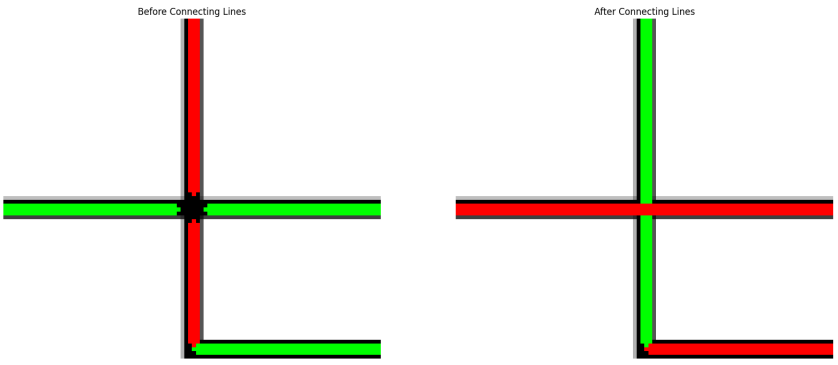


Figure 2.7: Intersection before and after connecting the line segments.

Converting the line segments into polylines at this stage would produce unusable data, as the segments are not grouped and not sorted in the sequential order of the edge's *flow*. To generate proper polylines, the line segments must first be sorted into the correct order **??**. For example, if the first line segment in the list is an intermediate line segment within an edge, the polyline function would incorrectly attempt to connect the intermediate points directly to the endpoints, leading to incorrect detections.
The implemented method begins by selecting the first line segment, adding it as a starting point to the first chain and to a list of used segments and setting the chain_growing flag to true. It then iterates through the remaining segments, checking whether each segment has already been used and whether any of its points lie within 7 pixels of the endpoints of the current segment. If a match is found, the segment is added to the used_segments list and the current chain. If no segment is found within the 7-pixel threshold, the chain_growing flag is set to false, and the completed chain is added to the list of chains. This process continues until all line segments have been assigned to a chain 2.8

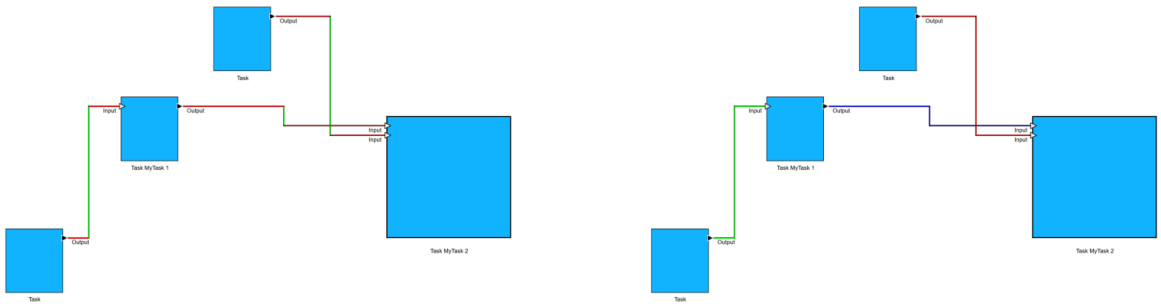


Figure 2.8: Left: detected vertical and horizontal line segments
Right: detected line segments processed into colored line segment chains

Generating the polylines now would yield better results, but still create unusable data, because the line segments within each chain and the two points within each line segment are not sorted. The first point in a list of line segments in a chain could for example be a point in the middle of the chain, resulting in OpenCV's Polyline function to connect the following points in the wrong order.
The sorting algorithm for solving this problem has two parts: the first sorts the line segments from beginning of the chain to the end, the second sorts the end- and startpoint of each line segment individually, so that they too appear in order of 'flow' in the chain.

To differentiate between intermediate points and endpoints the function utilizes the way in which the points where found in the first place: when two line segments intersect to form a 90Âř degree turn, each segment consists of a start- and an endpoint. This means, there are always two points in close proximity at intermediate points, where line segments meet, but only one point at the endpoints, because there is only one line segment ending at each endpoint **??**. Using this discrepancy, the function identifies endpoints by iterating through all points and checking for any other points within a five-pixel radius. Any point without any other points within this radius is considered an endpoint of a chain.

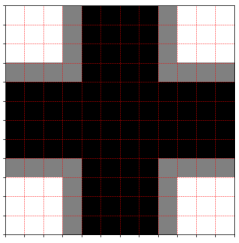


Figure 2.6

Using the identified endpoints to initialize the sorted_chain list allows the program to organize the chains systematically. The process begins by selecting the segment that includes one of the start points as the initial segment of the chain. The endpoint of this segment that is not a chain endpoint is designated as the first last_point in the sorted_chain list. To determine the next segment in the chain, a lambda function calculates the distance

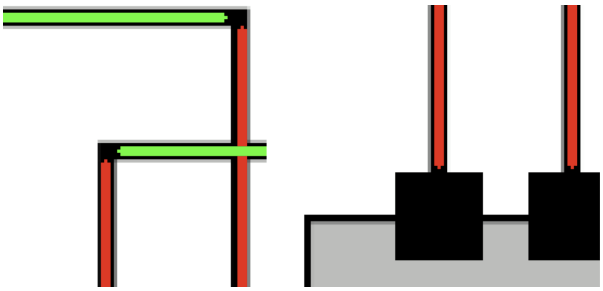


Figure 2.9: Difference between intermediate points and endpoints

between the last point and both points of each remaining segment. The segment containing the point with the smallest distance to the last_point is selected as next_segment and removed from the list of remaining_segments. Within this segment, the point closest to the last_point is appended first to the sorted_chain, followed by the other point of the segment. This process repeats until no segments remain in the remaining_segments list. The entire process is repeated for each chain until all points in all chains are ordered according to the edge's flow.

To generate polylines, ordered lists of points, from the sorted chains, the method iterates through the sorted_chains list, appending each point in sequence. Once the polyline is constructed, it is converted into the format required by OpenCV for further processing.

2.1.2 Further Improvements

This method successfully detects and processes nearly all edges within XGEE. In contrast to the previous algorithm, it can identify edges in any orientation, regardless of their start or endpoint or order of their line segments. Furthermore, it is capable of processing and interpreting intersections and signal containers, making it well-suited for handling large, complex models. The method performs reliably across all three editor modes, provided the models are formatted correctly. A notable problem arises when vertices with surrounding black edges are scaled sufficiently, making their edges resemble signal-carrying edges, which complicates their differentiation. In this case, a possible solution would be to let the vertex detection run first and exclude the found areas when applying the edge detection. Alternatively, the presence of colored pixels around the edges could be checked, as the background around edges is typically white, unlike the areas inside device or function vertices. This way, any obstructions caused by the vertices can be avoided.
Another issue may arise if the edges are configured by the user in an unexpected way. For instance, intersection within a few pixels of each other or intersections hidden behind vertices cannot be properly interpreted by the current method, which could lead to unexpected results. Future improvements of the XGEE editor, such as a more advanced automatic arrangement algorithm, could reduce or eliminate the risk of ambiguous user input. Additionally, enhancing the readability of subtask edges within the allocations editor would be beneficial for improving the verification tool. Currently, these edges often overlap with text, are thin and have poor contrast, making them difficult for the current edge detection pipeline to detect accurately.

2.2 Vertex Detection

Vertices represent distinct visual elements within XGEE such as functions, devices, containers or IO ports. Identifying these vertices is critical for interpreting the structural arrangement of diagrams. This section introduces a template-matching approach to address problems including overlapping vertices and varying sizes, ensuring a more reliable vertex detection in all relevant

diagram types within XGEE.

A major problem in template matching is the differing occurances of the searched for template in the images. Especially in the allocations editor, there are blue subtasks inside the grey device boxes and blue arrows inside the container. Each of these vertices "contaminates" the underlying vertex and lowers the number of matching pixels in that area. To combat this issue, the subtasks, which are not obscured by other vertices and cannot be resized, are found first using regular template matching. The found bounding boxes are then covered using the same grey color as the surrounding device boxes. This ensures the detection of the devices in the following step works more reliable, independent of the number of subtasks inside the device.
for the containers, this approach would work as well, however there is a simpler solution. Because the signal arrows and their attached names tend to be placed in the center of the container, a mask can be used when template matching, to only consider the outer edge of pixels of the container. As long as it has a color different of its surroundings, this approach supplies a simmilar amount of reliablity as the solution used for the subtask detection.

2.3 Text Recognition

The original text detection in XGEE utilized Pytesseract, an open-source optical character recognition (OCR) engine developed and sponsored by Google in 2006. Pytesseract required separate installation from other packages and could only detect text in images that had been preprocessed. Additionally, the output data required extensive postprocessing to become usable. While Pytesseract demonstrated high accuracy and speed, it struggled to reliably detect small text or text with low contrast to the background. Its optimization for structured text formats, such as those found in books, further hindered its performance in XGEE, where text can appear in varying orientations, sizes, and positions. These limitations made reliable text detection using Pytesseract difficult to achieve.

2.3.1 Methology

After evaluating various OCR engines, including EasyOCR [https://pypi.org/project/easyocr/], Doctr [https://pypi.org/project/python-doctr/], and Keras-OCR [https://pypi.org/project/keras-ocr/], EasyOCR proved to be the most suitable alternative. EasyOCR is a deep-learning-based OCR engine that reliably detects text in images, even under challenging conditions such as low contrast or resolution. Although it operates more slowly on modern CPUs compared to some alternatives, it performs significantly faster on GPUs. Additionally, its ability to detect text in multiple languages adds potential value for future applications. Integration of EasyOCR into the XGEE editor for the user is straightforward, as it can be installed directly via pip install easyocr through the requirements.txt file without requiring additional dependencies or downloads. Unlike Pytesseract, EasyOCR does not necessitate image preprocessing, and it structures found characters into words and sentences automatically based on proximity, eliminating the need for extensive postprocessing. These advantages make EasyOCR the optimal choice for the updated text detection pipeline within XGEE.

First, a reader object is created and the language of the text is specified. The readtext function of the reader object is then called with the image as an argument. The image has to be padded to have a square shape, so there are no problems rotating the data later. This function returns a list of tuples, containing the detected text, its bounding box and a certainty factor. This step is repeated with the rotated image. The positions of the bounding boxes of the found rotated text are then rotated around the center of the image to reallign them with the found text of the original image. Reading an image which contains rotated text usually results in

many falsely read characters, because for example 'o' and 'l' can be interpreted regardless of orientation. To filter out the falsely red characters, the algorithm first combines the found results of both ocr-searches and then removes every word shorter than three letters. Because easyocr automatically groups detected characters into words and sentences, this is a simple and effective method to filter out falsely read characters which is able to detect all text in the 10 testcases within this paper. Using this approach also allows the same algorithm to be used for all images, regardless of the XGEE editor they were created in.

2.3.2 Future Improvements

In the future, this method could be improved by using a more advanced algorithm to filter out falsely read characters. and checking the current editor to only search for rotated characters if they are actually present in the image.

3 Generalization

3.1 Edge Detection

3.2 Vertex Detection

3.3 Text Recognition

4 Implementation

5 Demonstration

5.1 Testcases

5.1.1 Testcases 1 - 3

5.1.2 Testcases 4 - 10

5.1.3 Testcases 11 - 15

5.2 Evaluation

6 Discussion

6.1 Feasability of Proof of Concept

6.2 further improvements

Bibliography

[1] . “Making your own linear filters! ”[Online]. Available: https://docs.opencv.org/3.4/d4/dbd/tutorial_filter_2d.html.

[2] S. Suzuki *et al.*, “Topological structural analysis of digitized binary images by border following,” *Computer Vision, Graphics, and Image Processing - 30(1):32âĂŞ46*, 1985.

[3] D. H. Douglas and T. K. Peucker, “Algorithms for the reduction of the number of points required to represent a digitized line or its caricature,” *Cartographica: The International Journal for Geographic Information and Geovisualization - 10 (2): 112âĂŞ122*, 1973.

[4] . “Matchtemplate() comparison methods. ”[Online]. Available: https://docs.opencv.org/3.4/df/dfb/group__imgproc__object.html#ga3a7850640f1fe1f58fe91a2d7583695d.