Bachelorarbeit

Visualization Verification of Complex Avionic Models Using Computer Vision

Franz Köhler

Zeitraum: 07.10.2024 – 07.02.2025

Betreuer: Andreas Waldvogel

Institut für Luftfahrtsysteme
Universität Stuttgart
Professor Dr.-Ing. Björn Annighöfer
Professorin Dr.-Ing. Zamira Daw

Nr. B-108

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

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.

Task
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

Date, signature student:_____

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.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit / Masterarbeit selbständig mit Unterstützung 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 erkläre weiterhin, bei der Erstellung der Arbeit die einschlägigen Bestimmungen zum Urheberschutz fremder Beiträge entsprechend den Regeln guter wissenschaftlicher Praxis[1] eingehalten zu haben. Soweit meine Arbeit fremde Beiträge (z.B. Bilder, Zeichnungen , Testpassagen etc.) enthält, habe ich diese Beiträge als solche gekennzeichnet (Zitat, Quellenangaben) und eventuell erforderlich gewordenen Zustimmungen der Urheber zu Nutzung dieser Beiträge 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 Köhler

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

Nutzungsrechterklärung

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

Visualization Verification of Complex Avionic Models Using Computer Vision

in der Institutsbibliothek des Institutes für Luftfahrtsysteme mit sofortiger Wirkung öffentlich zugänglich aufbewahrt und die Arbeit auf der Institutswebseite sowie im Online-Katalog der Universitätsbibliothek 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 Prüfexemplar eine weitere gedruckte sowie eine digitale Fassung übergeben.

Der Universität Stuttgart übertrage ich das Eigentum an diesen zusätzlichen Fassungen und räume dem Institut für Luftfahrtsysteme an dieser Arbeit und an den im Rahmen dieser Arbeit von mir erzeugten Arbeitsergebnissen ein kostenloses, zeitlich und örtlich unbeschränktes, einfaches Nutzungsrecht für Zwecke der Forschung und der Lehre ein. Falls in Zusammenhang mit der Arbeit Nutzungsrechtsvereinbarungen des Instituts mit Dritten bestehen, gelten diese Vereinbarungen auch für die im Rahmen dieser Arbeit entstandenen Arbeitsergebnisse.

Stuttgart, den 07.10.2024

Franz Köhler

Abstract

Visualization Verification of Complex Avionic Models Using Computer Vision

With the growing complexity of applications in aviation, the use of Domain-specific Modeling (DSM) in the field has become vital. It enables engineers to work on larger and more complex applications 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 verification effort. One important aspect of DSM in these applications is ensuring a correct model visualization to increase safety and reduce the amount of manual verification work.
This thesis aims to improve the reliability of the automated verification of block-diagram visualizations in DSM. Computer vision techniques are 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.
This thesis extends the capabilities of the block diagram recognition algorithm to work with complex and diverse diagrams in three graphical Domain-specific Languages (DSLs). 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.
The new implementation is evaluated using a set of 20 unique test cases, each containing one or two block diagrams with a single simulated error. The results show that the implementation is able to correctly identify and textually indicate all simulated errors to the user, differentiating between different error types. If possible, the implementation visually provides the position of the found errors inside the model editor alongside the textual indication.

Kurzzusammenfassung

Verifikation von Visualisierungen von komplexen Avionik Modellen mit Computer Vision

Mit der steigenden Komplexität von Anwendungen in der Luftfahrt wird die Nutzung von Domain-specific Modeling (DSM) in diesem Bereich immer wichtiger. Es ermöglicht Ingenieuren, effizienter an größeren und komplexeren Anwendungen zu arbeiten und reduziert durch automatische Code-Generierung die Anzahl der Fehler in den resultierenden Programmen. Bei sicherheitskritischen Anwendungen jedoch ist DSM durch die nötige Verifikation der Modell-Visualisierungen mit signifikantem Mehraufwand verbunden.
Diese Arbeit zielt darauf ab, die Zuverlässigkeit der automatisierten Verifikation von Blockdiagramm-Visualisierungen in DSM zu verbessern. Techniken der Computer-Vision werden verwendet, um Blockdiagramm-Modelle zu erkennen und zu verarbeiten. Die erkannten Daten werden mit dem ursprünglichen Modell verglichen, um Abweichungen zu finden und dem Benutzer innerhalb eines browserbasierten grafischen Modelleditors anzuzeigen.
Diese Arbeit erweitert die bestehende Implementierung der Blockdiagramm-Erkennung, um mit komplexen und vielfältigen Diagrammen in drei grafischen Domain-specific Languages (DSLs) zu arbeiten. Die neue Implementierung verwendet eine Kombination von Methoden aus der Computer-Vision, um Kreuzende oder teilweise verdeckte Verbindungslinien in verschiedenen Ausrichtungen, diverse Vertices in unterschiedlichen Größen und Anordnungen sowie Textblöcke in unterschiedlichen Orientierungen zu erkennen.
Diese Verbesserungen demonstrieren das Potenzial von Computer-Vision-Methoden, die Verifikation von DSM-Modellen zu automatisieren und die Sicherheit in sicherheitskritischen Anwendungen der Luftfahrt zu erhöhen.
Die neue Implementierung wird anhand einer Reihe von 20 Testfällen evaluiert, die jeweils ein oder zwei Blockdiagramme mit einem simulierten Fehler enthalten. Die Ergebnisse zeigen, dass die Implementierung in der Lage ist, alle simulierten Fehler korrekt zu identifizieren und textuell anzuzeigen, wobei zwischen verschiedenen Fehlertypen unterschieden wird. Wenn möglich, gibt die Implementierung zusätzlich zu der textuellen Anzeige auch die Position der gefundenen Fehler im Modelleditor visuell aus.

Contents

List of Figures

List of Tables

Index of abbreviations

API	Application Programming Interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DSL	Domain-specific Language
DSM	Domain-specific Modeling
EAST	Efficient and Accurate Scene Text Detector
EMF	Eclipse Modeling Framework
EMOF	Essential Meta-object Facility
GPU	Graphics Processing Unit
IO	Input-output
OAAM	Open Avionics Architecture Model
OCR	Optical Character Recognition
OOP	Object-oriented Programming
OpenCV	Open Source Computer Vision Library
R-CNN	Region-based Convolutional Neural Network
SLD	Single-line Diagrams
UI	User Interface
UML	Unified Modeling Language
XGEE	eXtensible Graphical EMOF Editor

1 Introduction

Development of complex, safety-critical systems requires an efficient way for engineers to ensure the correctness and reliability of the developed software. Domain-specific Modeling (DSM) provides an approach to achieve this by enabling the creation of models that are closely aligned with the specific concepts and requirements of a particular domain. This approach usually includes automatic code generation, which significantly reduces the risk of faulty applications [1].
By using DSM, engineers can work more effectively, as it allows for the abstraction of complex system details into more manageable and understandable representations. This not only enhances productivity but also the overall quality and safety of the developed systems.
Models can be represented visually using block-diagrams to further simplify the development process. Furthermore, DSM enables the reuse of domain-specific knowledge and components.
In safety-critical domains, such as avionics, the ability to visualize models and automatically generate code from them ensures that the software adheres to safety standards and reduces the likelihood of human errors during the development process.

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 an 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 verification tool for use in DSM [2].

A typical use case for DSM in avionics could be a door opening system. The system consists of a door, a motor, a sensor and a control unit. The door can be opened and closed by the motor, which is controlled by the control unit. The sensor detects whether the door is open or closed. The control unit receives the sensor data and controls the motor accordingly.
This system can be modeled using a block diagram, where the door, motor, sensors and control unit are represented as blocks, and the connections between them are represented as lines. Another diagram can be used to model the airplanes hardware components and their connections, such as the core processing units, remote data concentrators, sensors and actuators. A third diagram can be used to allocate the functions to the hardware components, as well as the connections between them.
This diagram-based approach allows users to visualize the system and its components, making it easier to understand and communicate. Diagram-based model editors include well established tools such as *Simulink* and *Enterprise Architect*, which are widely used in the industry.

2 Fundamentals

The following chapter provides an overview of the fundamentals of the technologies and concepts used in this thesis. It covers the eXtensible Graphical EMOF Editor (XGEE), computer vision, Domain-specific Modeling (DSM), the challenges in XGEE's visualization verification and the state of the art in automatic diagram interpretation.

2.1 eXtensible Graphical EMOF Editor (XGEE)

XGEE is a graphical model editor, which is currently under development at the Institute of Aircraft Systems. This editor is designed to facilitate the creation and modification of ecore-based models through a browser-based graphical user interface. Generally, XGEE can be used to work with any kind of model, but this thesis focuses on its use in aviation.
The primary objective of XGEE is to provide a user-friendly platform that allows engineers to efficiently design and visualize complex avionic systems, possibly working simultaneously on the same model. A demonstrator of XGEE is currently available online.[1]
Within XGEE, three distinct types of *tokens* are utilized across three specialized editors to represent various components of avionic models. These editors are:

- signals
- vertices
- text labels

We consider XGEE editors for three layers of the Open Avionics Architecture Model (OAAM): the functions editor, the hardware editor and the allocations editor. OAAM supports additional layers such as the *restrictions layer* and *capabilities layer*, which are not considered in this thesis. In *XGEE*, each of the three editor models utilizes a unique set of *.svg* files to represent different components within the avionic model. These editor models are the *functions editor*, the *hardware editor*, and the *allocations editor*. An overview of these editors is provided in the following sections.

2.1.1 Functions editor

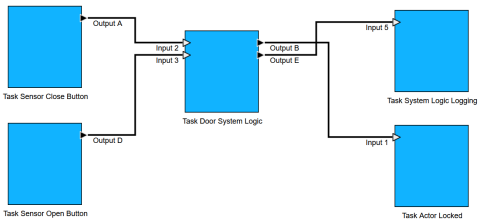


Figure 2.1: Example of a diagram in the Functions editor.

The functions editor is used to define avionic functions and their interactions. Functions are represented as large blue boxes, with their interactions shown through black signals connecting inputs and outputs. Inputs and outputs are depicted as small black-and-white triangles located on the left and right edges of the function boxes. Inputs, outputs and functions have visible text labels (see Figure 2.1).

[1] `https://xgee.de/en/`

2.1.2 Hardware editor

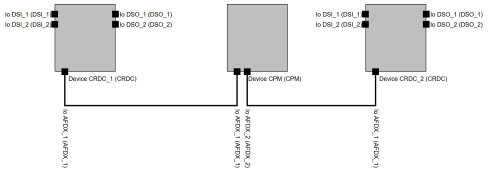


Figure 2.2: Example of a diagram in the Hardware editor.

The hardware editor is used to define hardware components and their physical connections. Hardware components are represented as large gray boxes, and their connections are illustrated as black signals linking Input-output (IO) ports. These IO ports appear as small black squares positioned along any edge of the hardware boxes. Like in the functions editor, IOs and devices have visible text labels (see Figure 2.2).

2.1.3 Allocations editor

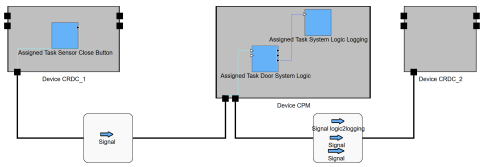


Figure 2.3: Example of a diagram in the Allocations editor.

The allocations editor is used to map functions to hardware components. Visually, it is simmilar to the hardware editor, with allocated functions represented as small blue boxes nested within larger gray hardware boxes. The specific signal transmissions between the hardware components are illustrated as white signal containers that overlap with the corresponding physical connections. These boxes contain the signals being transmitted through the corresponding connection. Inside the gray hardware boxes, connections between functions and IO's, are illustrated as thin, color-coded signals. They represent the same connections proviously defined in the functions editor, but are now allocated to specific hardware components (see Figure 2.3).

This thesis builds upon the work of Andreas Waldvogel and Björn Annighöfer in [2] to further automate the verification process within XGEE by tokenizing a screenshot of the editor window. This means detecting the bounding boxes and token types of all elements of the diagram. To recognize and process the screenshot data, methods from the Python library *OpenCV* are being used.
The complete process of converting a screenshot of a block-diagram into meaningful error indications requires a series of individual steps, as shown in Figure 2.4.
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, including issues such as 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, or blocks obscuring other blocks.
Fundamentally, this verification approach can be applied to any model-based application like Simulink. However, implementing the model comparison step would require much work, as the model has to be reconstructed from the recognized tokens, which is not trivial.
The goal of this thesis is to provide a tool that can be used to verify the correctness of the visualization of avionic models within XGEE's functions and hardware editors, enabling engineers to work more effectively and with a higher degree of confidence.

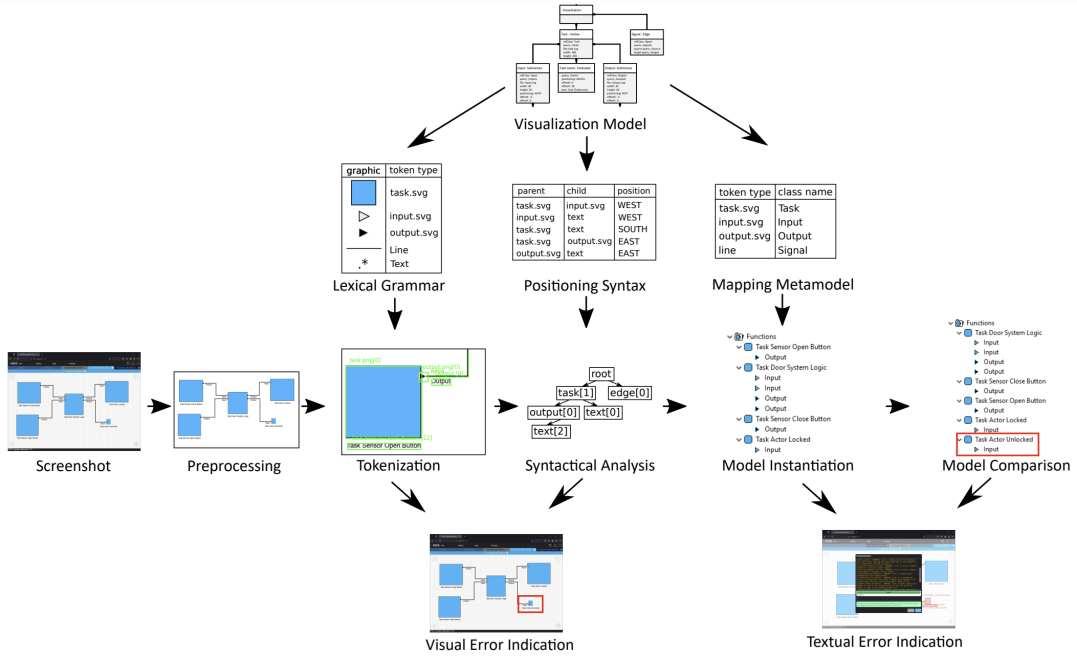


Figure 2.4: Steps in the visualization verification process [2].

2.2 Computer Vision

Humans have the remarkable ability to efficiently perceive and interpret visual information, extracting meaningful insights from their surroundings with ease. Tasks such as recognizing familiar faces, estimating distances, or identifying irregularities in a road surface may appear trivial to us, yet they present a significant challenge for computers to replicate.
Computer Vision is the field of mathematical models and approaches that enable computers to recover, interpret, and understand information from images or videos like humans. It encompasses a wide range of tasks, including image recognition, object detection, automation and more. It has numerous applications in a variety of fields, such as autonomous vehicles, facial recognition and medical imaging. Eventhough the field has made significant progress in recent years, many challenges such as handeling complex environments in real-time, detecting objects in low-light conditions or interpreting ambiguous data remain unsolved.
In avionic model development, visual representations of models are used to communicate complex systems and their interactions. To ensure correctness of these models, computer vision tools are used to interpret and verify the visual model representations, eliminating the need for manual verification.

2.3 Domain-specific Modeling

When developing safety-critical real-world applications, such as an avionics system, Object-oriented Programming (OOP) enables developers to create complex systems by defining classes and objects that interact with each other. However, as the complexity of these systems increases, it becomes harder for many developers to collaborate on the same project, as they need to understand the entire system to make changes.

The Unified Modeling Language (UML) is a generic and source-code-independent OOP description language that can be used to model software systems. It provides a standardized way to visualize the design of a system using UML diagrams, making it easier to understand and communicate. These diagrams consist of clearly defined elements specified in the UML standard and can be directly converted to code through automatic code generation.

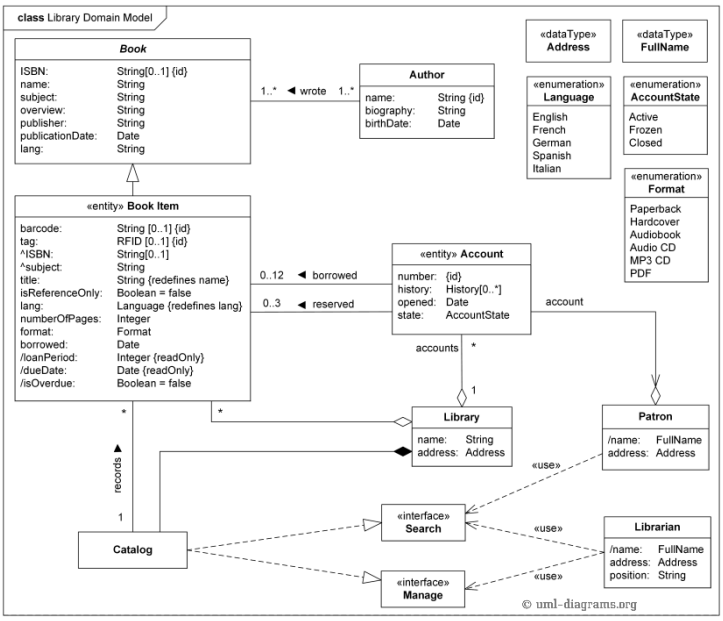


Figure 2.5: Example of an UML class diagram describing a library management system. In highly specialized domains, such as avionics, an UML diagram might be too abstract to clearly represent the system. [3]

Figure 2.5 shows an example UML class diagram describing a library management system. While UML provides a standardized approach for modeling software systems in a general-purpose manner, it may not fully address the specific needs of highly specialized domains, such as avionics. Domain-Specific Modeling (DSM) addresses this issue by enabling engineers to create models that are closely aligned with the concepts and requirements of a particular domain. For example, in avionics systems, DSM might use specialized notation to represent specific aircraft components, such as sensors, actuators, or flight control systems, rather than relying on abstract classes and objects like UML. This reduces the semantic gap between the model and the real-world implementation.
In most cases, a Domain-specific Language (DSL) is developed by a small group of experts within a company or within a collaboration between companies and tailored to their unique needs, then used consistently throughout the organization to ensure uniformity and efficiency. This approach simplifies design processes and ensures that models are more easily validated, improving both reliability and safety, both critical requirements in the development of avionics systems.

2.4 Challenges in XGEE's visualization verification

The XGEE editor is a browser-based model editor that, in our application, allows users to create and edit avionic models using a graphical interface but in general, XGEE can be used to edit anything that uses an ecore metamodel. For this thesis, we conside three editor models: the functions editor, the hardware editor and the allocations editor, each useing a different set of tokens to represent different elements of the model.
To verify the correctness of this visualization, a screenshot of the model is tokenized, meaning that the bounding boxes and token types of all elements of the diagram are detected. The recognized tokens are checked for correct syntax and unrecognized pixels. Then they are used to rebuild the model and compare it to the original, highlighting any visualization errors.
The primary challenge addressed in this paper is the accurate detection of these tokens. This includes handling intersecting and overlapping signals, obscured vertices and text labels, as well

as large and complex models.
Furthermore, the integration of new methods for token detection into the existing codebase introduces an additional layer of complexity. Another challenge adressed in this thesis is to make the verification more versatile by enabeling it to change with the model. This is achieved by making the verification *model-driven*, allowing the methods to query the current model for information and dynamically change their behavior based on it.

2.5 State of the Art

Automatic diagram interpretation has been a topic of interest in the field of computer vision and DSM. It could enable engineers to utilize diagrams that are currently only available as images or drawings, which would otherwise require manual reverse engineering. However, in large projects with consistent and well-maintained databases, most diagrams are already stored in usable formats, reducing the practical demand for automatic diagram interpretation in the industry. Consequently, no commercial tools for verified block diagram recognition are currently available on the market.

The work most similar to this thesis is presented in [4], which utilized a specifically trained Convolutional Neural Network to classify common symbols used in *Piping and Instrumentation Diagrams*, achieving an accuracy of 90%. They detected connecting lines using a graph search approach. By representing the pixels within the diagram image as a graph of black and white nodes, connecting lines can be identified by starting at any node corresponding to a symbol and traversing the diagram graph along its black nodes, keeping track of connected symbols along its path. For text detection, they used *EAST*, a text detectrion pipeline using a neural network. However, their method did not address the indication of uncertainties to the user. Instead, their primary goal was to digitize a database of diagrams to enable applications such as diagram search and machine learning-based predictive maintenance in the industry.
Recent research has focused on the recognition of handwritten diagrams, mathematical equations, flowcharts, and circuit diagrams. For example, [5] proposed a method for normalizing images captured by hand at arbitraty orientations to improve recognition accuracy and reliablity.
[6] used *Arrow R-CNN*, a deep-learning model and an extension of the *Region-based Convolutional Neural Network (R-CNN) object detector* [7] to detect and classify offline handwritten diagrams. *R-CNN* is an object detection framework to identify bounding boxes around objects and classify each object into its respective category. On a scanned flowchart dataset, the model achieved an accuracy of 78.6%, substantially improving the previous state of the art. However, their method was not designed recognize the diagrams structure or to adress the specific challenges of avionic model diagrams, particularly the need for validation and user feedback mechanisms.
Building on this work, [8] proposed *DrawnNet*, a CNN and keypoint-based detector capable of recognizing both symbols and diagram structures. Among other techniques, *DrawnNet* leverages arrow direction predictions to enhance diagram interpretation.
[9] proposed a framework for Single-line Diagrams (SLD) recognition, which are used in electrical engineering to represent power systems. Their methods include decomposing the diagram into seperate layers of electrical symbols and text labels to mitigate interference, using R-CNN to identify graphical symbols and the *super-resolution* technique to improve text label recognition.

The approaches discussed above face challenges due to the inherent ambiguity of loosely defined graphical modeling languages and non-ideal photos of diagrams. A significant amount of effort is typically spent on recognizing various styles, but they often prioritize maximizing detection without considering the level of confidence in the results. In contrast, our approach takes a different direction. By leveraging the precise definition of a graphical Domain-specific Language (DSL), we focus on a more structured verification process. Rather than attempting to recognize

all elements, our goal is to highlight areas where uncertainty exists in the visualization, providing more meaningful and targeted feedback. The methods proposed in this thesis are based on the work of [2], which introduced a method for tokenizing and validating a screenshot of an avionic model within the XGEE editor. The aim of this thesis is to extend this work by improving the accuracy of token detection and integrating new methods for more complex detection tasks.

3 Advanced Block Diagram Recognition

Accurately interpreting complex diagrams is essential for many analytical and computational tasks. In the context of XGEE, this involves breaking down screenshots into meaningful components through tokenization. This process, however, depends on the robustness and reliability of the underlying algorithms. This chapter aims to build upon the original algorithms, improving upon their detection accuracy, versatility and ensuring stability.

3.1 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.

3.1.1 Kernel-based Edge Detection

Since there are only vertical and horizontal edges within XGEE, two different kernels are used to find all pixels containing 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 vertical kernel in Figure 3.1 is rotated by 90° to generate the horizontal kernel.

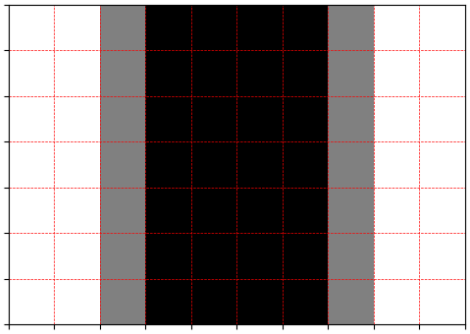


Figure 3.1: kernel used for vertical edge detection, rotated by 90° to generate the horizontal kernel.

The same process can be expressed using Equation 3.1, where H is the resulting matrix, I is the original image and K is the used kernel, with x, y, i and j representing individual pixels within the image and the 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 [10].
Currently, only a single line width defined by the kernel size is supported, which is enough for the functions and hardware editor. The allocations editor, however, will require support for multiple line widths and low contrast lines.
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 and processed further using OpenCV's *thresholding()* function to extract the detected pixels (see Figure 3.2 and 3.3).
As illustrated in Figure 3.4, ports and letters are sometimes misidentified as edges. Additionally, intersections of edges as well as 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.

$$H(x,y) = \sum_{i=0}^{M_i-1} \sum_{j=0}^{M_j-1} I(x+i-a_i, y+j-a_j) \cdot K(i,j) \tag{3.1}$$

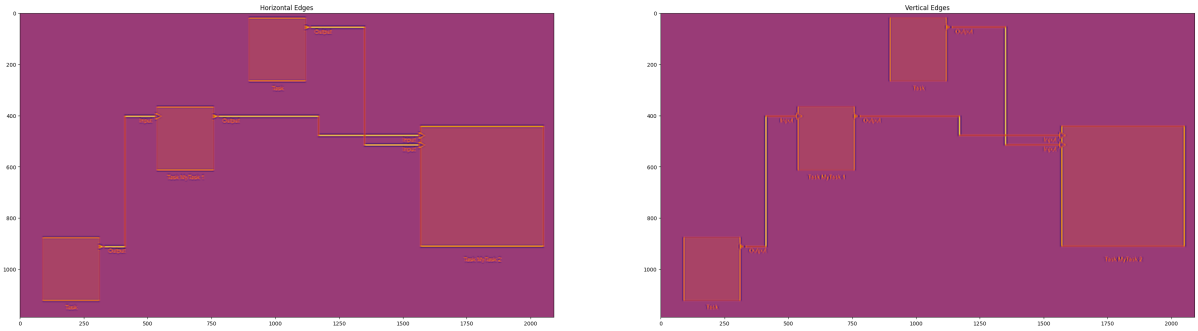


Figure 3.2: Filter2D function results for the horizontal (left) and vertical (right) kernel before thresholding. *Edge-like* pixels are colored more yellow, while *non-edge-like* pixels are colored more purple.

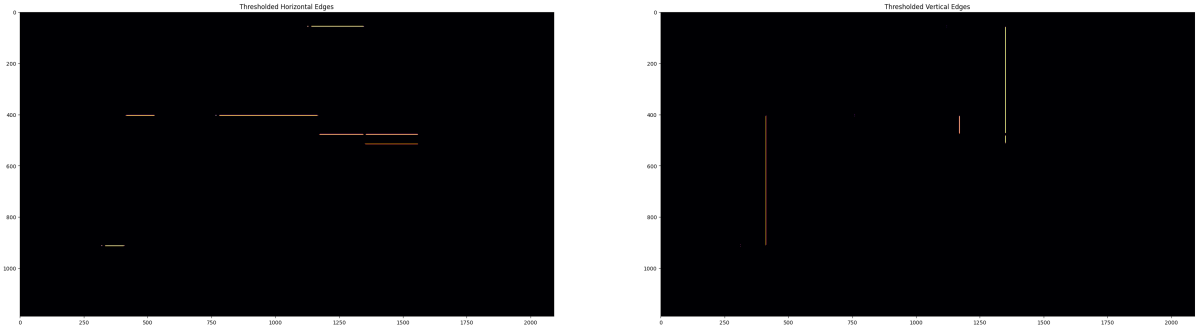


Figure 3.3: Thresholded filter2D function results. *Edge-like* pixels are now isolated.

For easier subsequent processing and data storage, the thresholded pixels seen in Figure 3.3 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 by Satoshi Suzuki and others in [11]. In this context, it is used to group nearby pixels and represent them as narrow polygons.
approxPolyDP(), which approximates a curve or a polygon with another curve or polygon with less vertices using an algorithm introduced by David H Douglas and Thomas K Peucker in their paper: [12]. This function is used to simplify the contours found by *findContours()* into line segments.

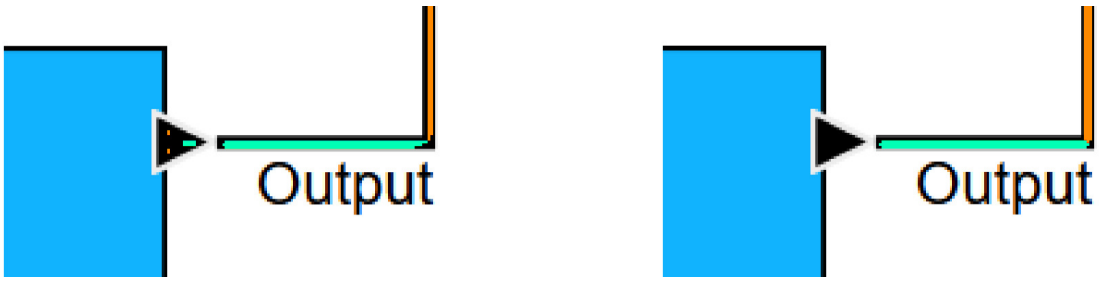


Figure 3.4: Comparison of detected pixels and line segments around an output before and after filtering short segments.

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, illustrated in Figure 3.4.

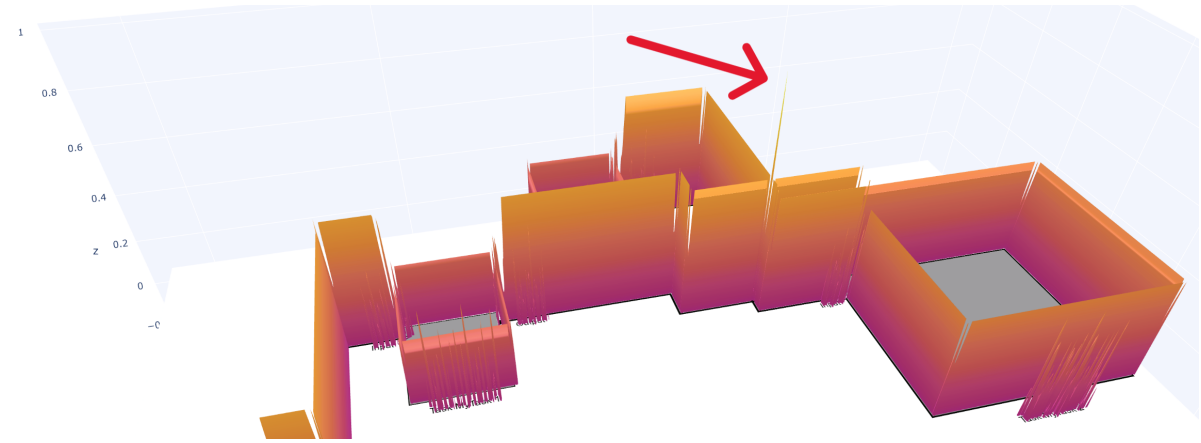


Figure 3.5: Similarity score peak at an intersection visualized in 3d using *plotly*. X and Y axes show the screenshot in grayscale, while the Z axis represents the similarity score.

3.1.2 Intersection Detection

As shown in Figure 3.3, 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 90° turns are considered impossible and will result in an error caused by the incorrect edge recognition, as it would result in an ambiguous diagram.
First, all intersections in the image are detected using OpenCV's *matchTemplate()* function, which matches a template image of an intersection, as seen in Figure 3.6, to overlapping regions of the target image [13].
The function slides the template across the image, comparing overlapping patches with the template using a specified method. Among the available methods [14], *Sum of square differences normed* (*tm_sqdiff_normed*) produced the most accurate results. For each pixel, according to Equation 3.2, the function calculates and assigns a value $R(x, y)$ representing the similarity between the template $T(x', y')$ and the corresponding image region $I(x + x', y + y')$ below. While this approach is more precise than *filter2D()*, it is also significantly more resource-intensive.

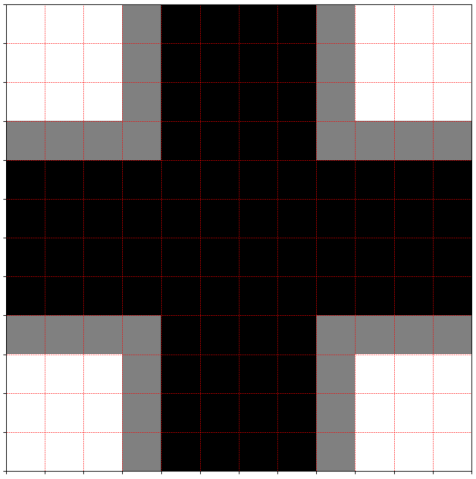


Figure 3.6: Template image used for intersection detection.

$$R(x, y) = \frac{\sum_{x',y'}(T(x', y') - I(x + x', y + y'))^2}{\sqrt{\sum_{x',y'} T(x', y')^2 \cdot \sum_{x',y'} I(x + x', y + y')^2}} \tag{3.2}$$

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 (see Figure 3.7) reveals that the white pixels near edges and text 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 3.5). 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 then processes each detected intersection by connecting the two vertical and two horizontal line segments adjacent to it. This eliminates any residual points near the intersection, leaving only one vertical and one horizontal line segment, as illustrated in Figure 3.9.

3.1.3 Container Detection

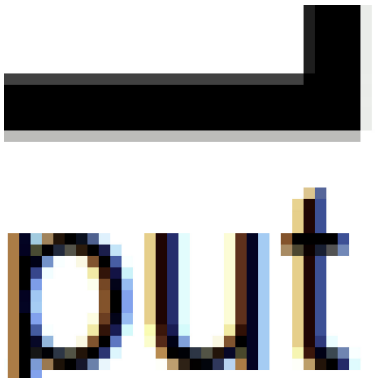


Figure 3.7: Aliasing typically found near hard edges to increase the apparent resolution or contrast. The shade of gray is not the same on both sides of the edge.

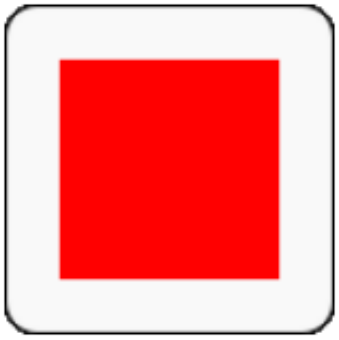


Figure 3.8: Template image used for signal container detection. The red area illustrates pixels ignored during template matching.

In the allocations editor, the same approach is applied to detect and process signal containers on top of edges. To enhance diagram readability, it is assumed that no 90° turns are concealed behind the containers, only one edge passes behind each container and that edges pass through them in a straight line, avoiding ambiguities that could confuse both computer vision algorithms and human users. 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.
The method detects all signal containers in the screenshot using template matching. To ignore any pixels in the center of the container template, a mask (illustrated in red in Figure 3.8) is used. This allows *signal arrow vertices*, indicating signals traveling through the underlying connection, to overlap with the container vertices, without causing any interference with the detection process. The detected containers are then processed in the same way as intersections, connecting the two line segments adjacent to the container.

Figure 3.9: Intersection before and after connecting the line segments.

3.1.4 Line Segment Grouping and Sorting

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 grouped into multiple lists of connected chains which then have to be sorted into the correct order. For instance, if the first line segment in the list is an intermediate segment within an edge, the polyline function may mistakenly attempt to connect its endpoints directly to the next point in the list, without considering whether it belongs to the same continuous chain, resulting in incorrect edge detections.
The implemented method begins by selecting the first line segment, adding it as a starting point to the first chain group and to a list of used segments and setting the *chain_growing* flag to true. It then iterates through all remaining segments, checking whether each segment has already been used and whether any of its points lie within 7 pixels of the points of the current segment. If a match is found, the segment is added to the *used_segments* list and the current chain group. If no segment is found within the 7-pixel threshold, the *chain_growing* flag is set to false, and the completed chain group is added to the list of chains (see Code Listing 3.1). A threshold of 7 pixels was selected as it reliably produces connected polylines while ensuring that closely positioned lines, such as those at ports, remain distinct and separate. This process continues until all line segments have been assigned to a group, illustrated with colors in Figure 3.10.

```
for segment1 in line_segments:
    if segment1 in used_segments:
        continue
    chain = [segment1]
    used_segments.append(segment1)
    chain_growing = True

    while chain_growing:
        chain_growing = False
        for segment2 in line_segments:
            if segment2 in used_segments:
                continue
            for chain_segment in chain:
                if (
                    distance(chain_segment.p1, segment2.p1) <= max_distance or
                    distance(chain_segment.p1, segment2.p2) <= max_distance or
                    distance(chain_segment.p2, segment2.p1) <= max_distance or
                    distance(chain_segment.p2, segment2.p2) <= max_distance
                ):
                    chain.append(segment2)
                    used_segments.append(segment2)
                    chain_growing = True
                    break
            if chain_growing:
                break
    chains.append(chain)
```

Code Listing 3.1: Grouping line segments into 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 consists of two steps: the first sorts the line segments from the 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 sequential order of 'flow' in the chain.

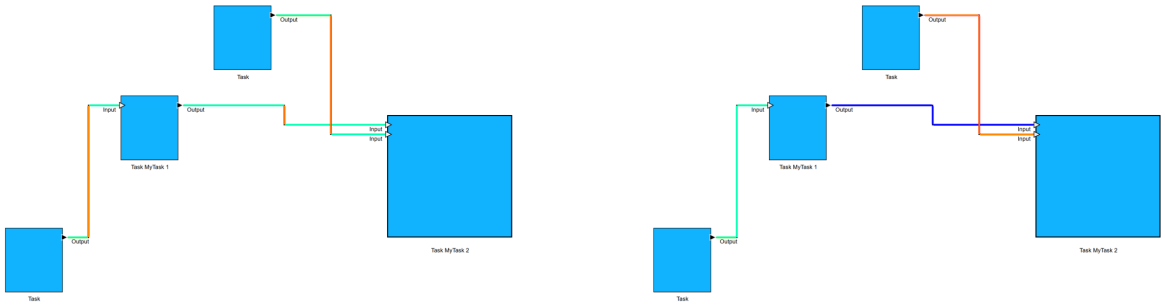


Figure 3.10: Left: detected vertical and horizontal line segments. Colors distinguishing horizontal and vertical lines.
Right: detected line segments grouped into individually colored line segment chains.

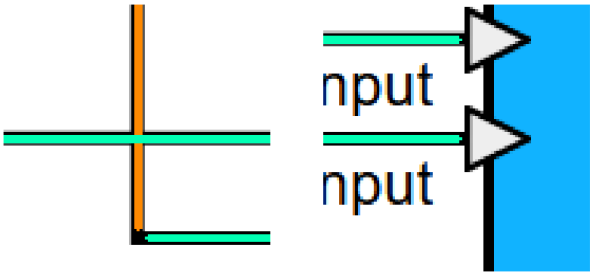


Figure 3.11: Intermediate points (left) have other points in close proximity, while endpoints (right) do not.

To distinguish between intermediate points and endpoints, the function relies on the method by which the points were initially identified: when two line segments intersect to form a 90° turn, each segment consists of a start- and an endpoint. As a result, intermediate points in a chain, where line segments meet, always have two points in close proximity, whereas endpoints only have a single point, since only one line segment terminates at each endpoint (see Figure 3.11). The method leverages this discrepancy by identifying endpoints through an iterative process: It examines each point and checks for the presence of other points within a seven-pixel radius. If no other points are found within this radius, the point is classified as an endpoint of a chain (see Code Listing 3.2).

```
chain_endpoints = []
for chain in chains:
    endpoints = []
    for segment in chain:
        for point in segment:
            if all(distance(point, other_point) > max_distance for
    other_segment in chain for other_point in other_segment if point !=
    other_point):
                endpoints.append(point)
    chain_endpoints.append(endpoints)
```

Code Listing 3.2: Differentiating between segment endpoints and intermediate points.

Using the identified endpoints to initialize the *sorted_chain* list allows the program to organize the chains systematically.
The process begins by selecting a 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* iterates through each remaining segment in the current chain, calculating the distance between the *last_point* and each point of each segment in the chain. 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 second 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* (see Code Listing 3.3).

```
sorted_chains = []
for index, chain in enumerate(chains):
    if not chain_endpoints[index]:
        sorted_chains.append(chain)
        continue
    start_point = chain_endpoints[index][0]
    sorted_chain = [start_point]
    remaining_segments = chain[:]

    while remaining_segments:
        last_point = sorted_chain[-1]
        next_segment = min(
            remaining_segments,
            key=lambda seg: min(distance(last_point, seg.p1), distance(last_point, seg.p2))
        )
        remaining_segments.remove(next_segment)
        if distance(last_point, next_segment.p1) < distance(last_point, next_segment.p2):
            if next_segment.p1 != last_point:
                sorted_chain.append(next_segment.p1)
            sorted_chain.append(next_segment.p2)
        else:
            if next_segment.p2 != last_point:
                sorted_chain.append(next_segment.p2)
            sorted_chain.append(next_segment.p1)
    sorted_chains.append(sorted_chain)
```

Code Listing 3.3: Sorting each point in each chain according to the edges *flow*

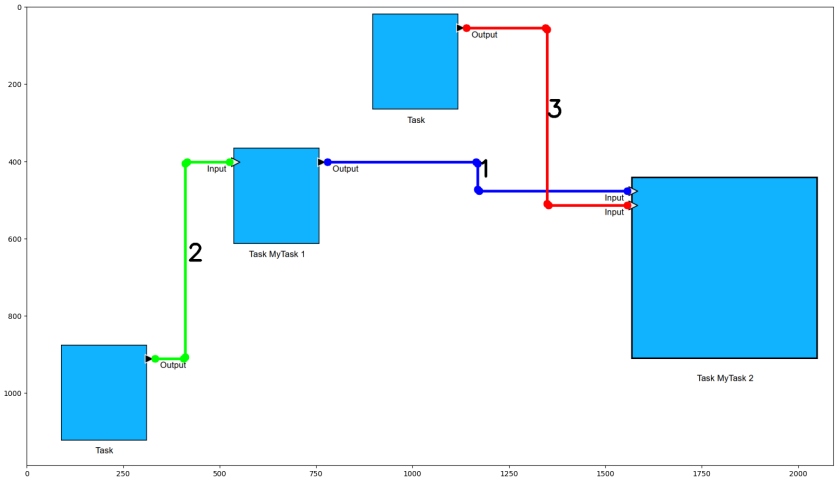


Figure 3.12: Numbered polylines found in the functions editor. Connecting the points in the correct order results in the detected edges.

As seen in Figure 3.12, the method successfully generates polylines from the initial screenshot. The order in which the polylines are initially detected depends on the order of the line segments

and is not relevant for the final result.
Polylines, which represent connected line segments as ordered lists of points, are a fundamental component in reconstructing edges within the diagram. To generate these polylines from the sorted chains, the method iterates through the *sorted_chains* list, sequentially appending each point to construct the polylines. The preprocessing and sorting of the data ensures that the chains are already structured correctly, making the generation of polylines straightforward and efficient. Once the polyline is constructed, it is converted into the format required by OpenCV for further processing.

3.2 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 the 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 considered diagram types within XGEE.

A major problem in template matching is the diversity of vertices within XGEE's editor models. For example, the functions editor contains only functions, inputs and outputs, while the hardware editor contains devices and IO ports and the allocations editor contains devices, signal containers, signal arrows and subtasks which overlap with devices and contain their own set of *sub-sub-vertices*. Additionally, some vertices are scalable, while others are not. Scalable vertices require extra processing to ensure accurate detection. However, applying this processing universally to all vertices would result in unpredictable and incorrect detections.

To address these challenges, the model is queried for a list of unique vertices. This list is analyzed to identify any vertices with the *isSubVertexBody*-flag set, which indicates they are subvertices such as subtasks, that overlap with other vertices. These subvertices are prioritized and placed at the beginning of the list of vertices. This way, the vertex detection method can progressively simplify the image by erasing detected subvertices after the first iteration of the vertex detection function, allowing the underlying vertices to be detected without obstruction. Other relevant flags are:

- *isScalable* which indicates whether the vertex can be scaled
- *sizeX* and *sizeY* which specify the size of the vertex
- *filepath* which specifies the path to the template image
- *parent_filepath* which specifies the path to the parent vertex in case it is a subvertex

The method iterates through the list of unique vertices in the current editor model, prioritizing subvertices identified in the previous step. These subvertices are removed from the image by setting all pixels within the bounding box of the subvertex to the *fill* color of the parent vertex as shown in Figure 3.13. This process effectively removes the subvertices from the image, enabling the subsequent vertex detection steps to accurately identify the remaining vertices.
After processing the subvertices, a new image object is created using the modified image. The method then iterates through the list of remaining vertices, loading the template image of each vertex using the *filepath* attribute. Depending on the *isScalable* flag, either the *ScalableVertexDetector* or one of the *UnscalableVertexDetector*s is initiated and the corresponding vertex detection is applied.

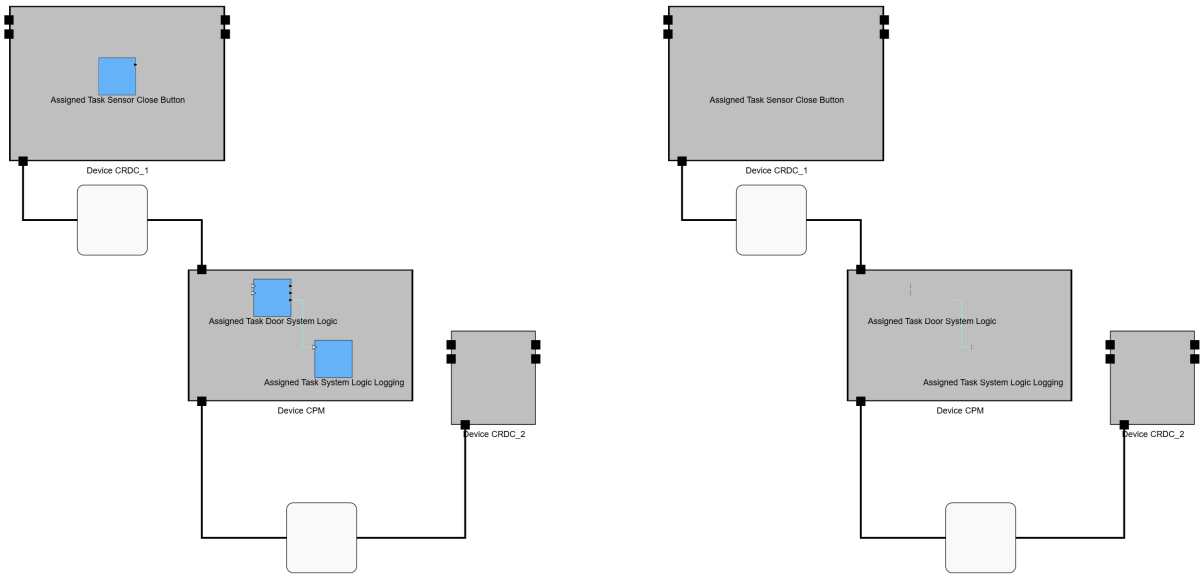


Figure 3.13: Original image from allocations editor containing subvertices (left) and modified image with subvertices removed (right).

In case of unscalable vertices, OpenCV's *matchTemplate* and *normalize* functions are applied. The resulting data is thresholded to identify the positions of all found vertices. To define the vertex boundaries, the *sizeX* and *sizeY* attributes are used, allowing bounding boxes of the correct size to be drawn with the detected match serving as the upper-left corner. This approach is effective for detecting vertices like IO ports and subtasks.
In case of scalable vertices, this approach does not work because the size of the vertex is variable, which reduces the certainty of found matches drastically and eliminates constant *sizeX* and *sizeY* values as a reliable method for defining bounding boxes.

Scalable vertices within XGEE are function- and device containers that can be resized to improve the readability of visualizations. However, the initial scalable vertex detection algorithm often produced inaccurate results when applied to large function- and device containers. Specifically, it tended to produce higher similarity scores at the corners of the container, as the black border surrounding the vertex at these locations more closely resembled the black border in the template image. In contrast, the absence of a black border in the center of the container led to lower similarity scores. In extreme cases, this resulted in the center of the vertex not being detected at all, with the algorithm instead falsely identifying four smaller vertices at the container's corners. The same issue occurred with large devices.
To detect these vertices regardless of their dimensions, multiple templates of the same size for different parts of the large vertex as illustrated in Figure 3.14 are required. They are generated from the (leftmost) original template image by cropping its edges and corners in different ways. Template matching is performed iteratively with each template, comparing the results after

Figure 3.14: Templates for scalable vertex detection generated from the original (leftmost) template image, black edges are exagerated for clarity.

each iteration. The highest similarity score for each pixel is retained and combined into the resulting data, which is then thresholded to extract all potential matches.

Typically, numerous matches are identified during this process. To eliminate false positives and process the matches to determine the differently sized bounding boxes of the vertices, the algorithm described by Andreas Waldvogel and Björn Annighöfer in [2] is applied.
The original image is thresholded to distinguish foreground from background pixels, as illustrated in Figure 3.15. This foreground information is then utilized to filter the detected matches, retaining only those located within the foreground area. This process effectively removes false positives which often occur because the black pixels along the edges and borders of function or device containers closely resemble those in the black borders in the template images.

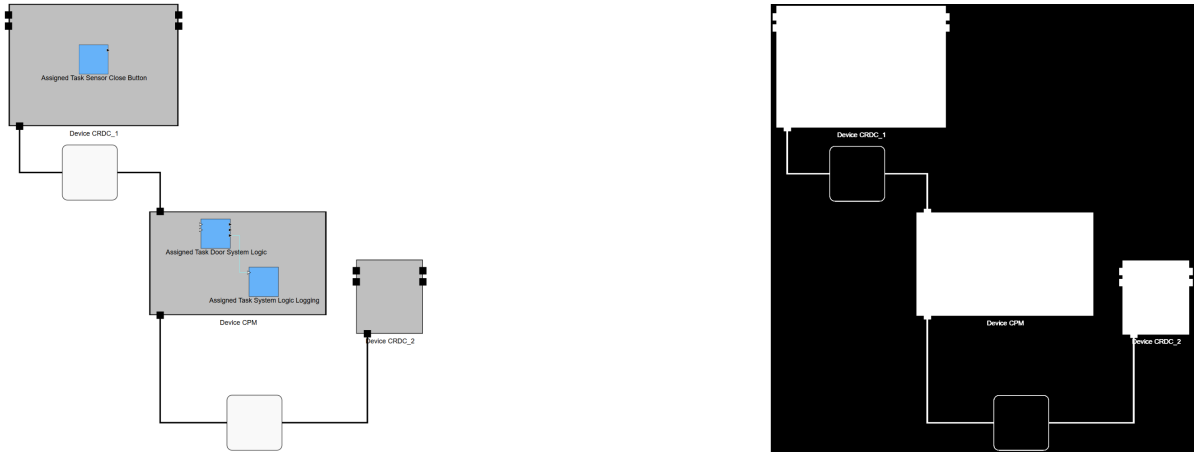


Figure 3.15: The allocations editor screenshot (left) is thresholded to extract the foreground pixels (right).

For each remaining detected match, the *sizeX* and *sizeY* attributes are used to create a filled bounding box around the detected position. Since multiple matches are often found at various locations within a single large vertex, the overlapping bounding boxes collectively form a larger structure. This structure is then simplified into a single bounding box using OpenCV's *findContours* function. This approach successfully detects all scalable vertices within XGEE

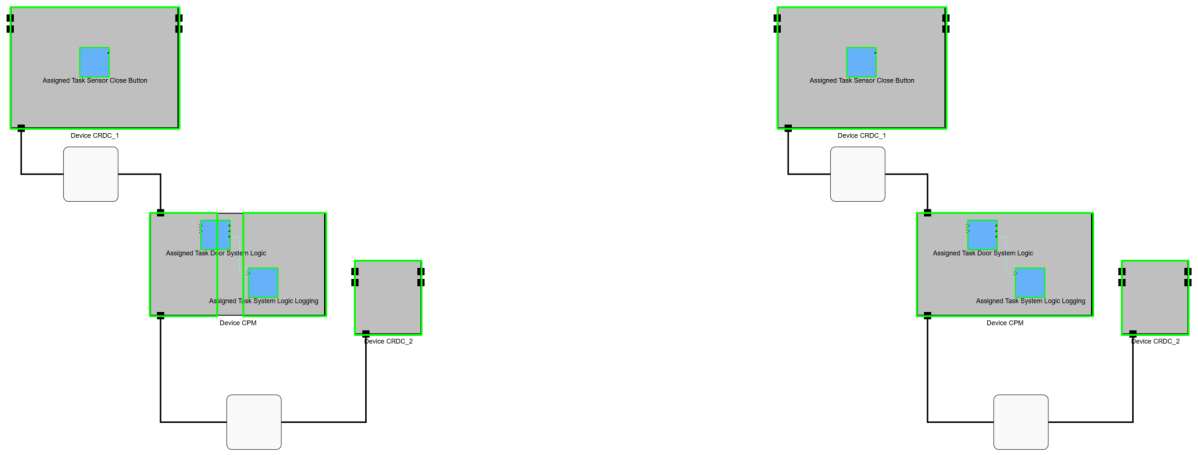


Figure 3.16: Image from allocations editor containing a large, partially detected device using one template(left) and containing a large, fully detected device using multiple templates(right).

regardless of their dimensions. Using both scalable and non-scalable vertex detection methods ensures that all vertices are accurately identified.
Figure 3.16 illustrates the original method's difficulties when dealing with large vertices with overlapping subvertices. On the left, in the center portion of the device, no matches are found

due to the absence of black borders and interference with the remaining text of the subtasks. On the right, many templates are used to detect the entire device, resulting in a more accurate detection.
The method is capable of detecting all vertices in the 20 unique test cases within this paper, including subvertices and scalable vertices.

3.3 Text Recognition

The original text detection in XGEE utilized Pytesseract [15], 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.

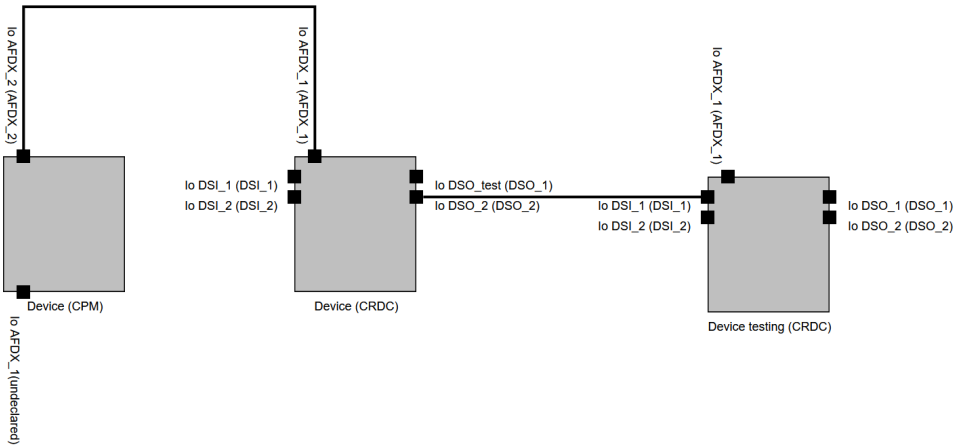


Figure 3.17: Input image from hardware editor containing rotated text

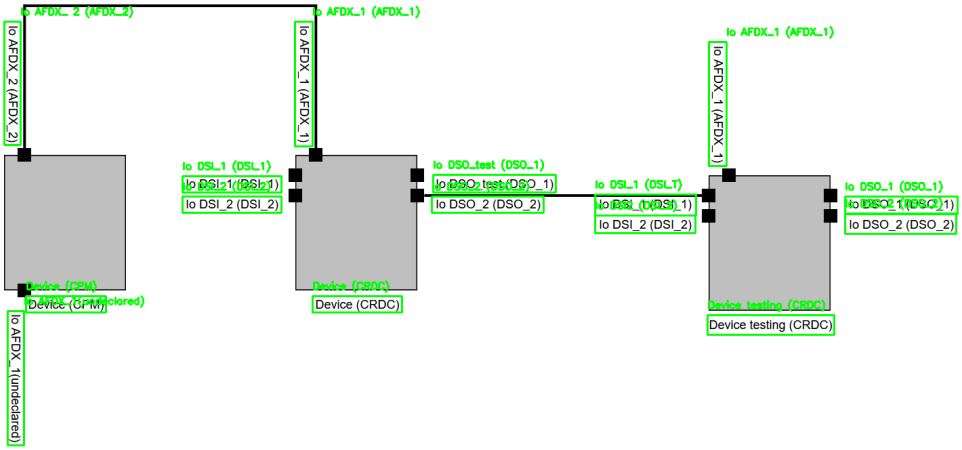


Figure 3.18: Output image from hardware editor containing text bounding boxes

After evaluating various OCR engines, including EasyOCR [16], Doctr [17], and Keras-OCR [18], 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 only necessitates minimal 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 it can be easily rotated. To ensure a more reliable detection, the image is slightly blurred and upscaled to counter any aliasing and small characters. The text detection function then returns a list containing the detected text, its bounding box, and a certainty factor. Parameters can be specified when calling this function to adjust the expected text properties, enhancing the reliability of the detection. For example in earlier versions, EasyOCR struggled to detect single numbers, likely because its language model has been trained on data not containing any single characters, but with these parameter adjustments, the occurance of this issue has been reduced.
The detection process is repeated for the rotated version of the image to detect rotated text labels, present in the hardware layer, as well. 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. The found bounding boxes and text are illustrated in Figure 3.17 and 3.18. 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 read characters, the algorithm first combines the found results of both OCR-searches and then removes every word shorter than three letters.

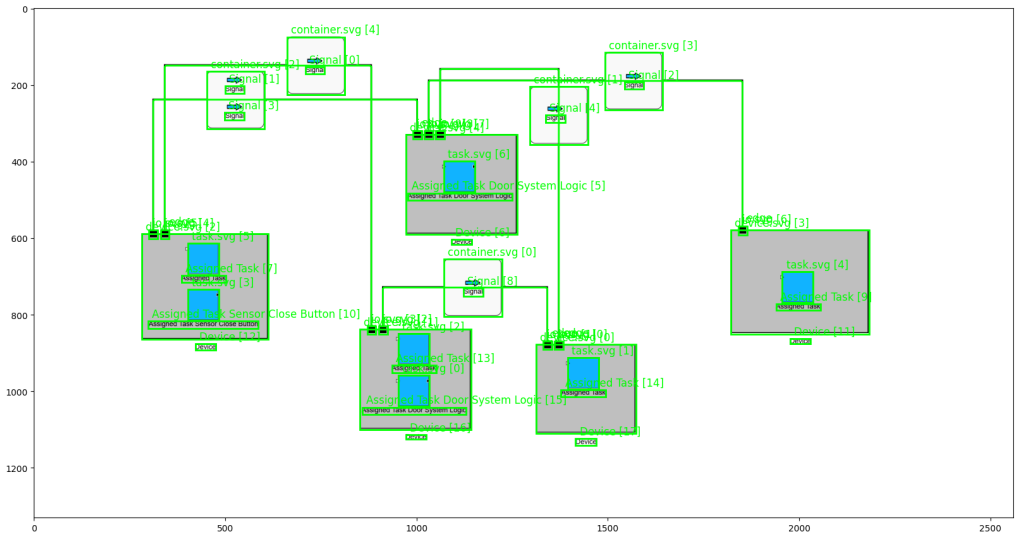


Figure 3.19: Debugging image of bounding boxes and token names found during the tokenization of the allocations layer, demonstrating the capability of the tokenization pipeline.

Because easyOCR automatically groups detected characters into words and sentences, this is a simple and effective method to filter out falsely read characters. Using this approach also allows

the same algorithm to be used for all models, regardless of the XGEE editor they were created in, simplifying the integration into the XGEE codebase. In conclusion, these methods enhance the tokenization process compared to the previous implementation. As seen in Figure 3.19, the combined methods are able to correctly identify all token types in the allocations editor, utilizing all previously mentioned methods.
The new edge detection identifies edges regardless of length and orientation. Intersections and signal containers are correctly identified and processed, ensuring that all detected edges are continuous. The vertex detection method is capable of detecting scalable vertices of varying sizes, unscalable vertices, IO ports, and subtasks, even when they overlap. The text detection method is capable of detecting text in varying orientations and positions. The combined methods provide a robust and reliable tokenization pipeline for XGEE, capable of accurately identifying all relevant tokens in the editor models.[1]

[1]The described methods can be found individually at `https://github.com/franzbanz/computervision` and implemented into XGEE at `https://gitlab.com/xgee/xgee-example-app`.

4 Integration

This chapter provides an in-depth explanation of how the described methods were implemented into the larger XGEE codebase.

4.1 System Overview and Workflow

XGEE is a web-based graphical model editor that utilizes an editor model to define editors for ecore models in a model-driven approach, enabling visualization and interaction. The editor model supports the model-driven definition of editors for ecore models, incorporating both visualization and interaction [2].

XGEE's visualization verification is implemented as a modification to the main editor, mostly within the *detector.py* and *diagram_tokenization_orchestrator.py* files. For the verification process to function correctly, a screenshot of the entire screen is preprocessed to isolate the relevant portions of XGEE's editor window by removing elements like the browser UI and background grid. During the tokenization step, the positions, dimensions and contents of edges, vertices, and text are detected and stored. This data is subsequently used in the syntactical analysis to identify inconsistencies in parent-child relationships between tokens. (Syntax in this context means the positioning rules of vertices, edges and text labels defined in the editor model.)
A new model is then instantiated based on the detected tokens and the visualization model. This model is compared to the original and any discrepancies are highlighted through both graphical and textual user interfaces.
Edge, vertex, and text detections occur during the tokenization phase of the verification pipeline. In the original integration, each detection algorithm was implemented as a class within the *detector.py* file, while their execution was managed by the *diagram_tokenization_orchestrator.py* file, which determined the order and process of the tokenization. This structure is retained and extended in the new implementation.

4.2 Edge Detection Integration

The integration of the new edge detection algorithm into XGEE is simplified by its use of the already existing interface for inputs and outputs. This approach ensures that the system remains modular and scalable.
To implement the new functionality, the algorithm is divided into sections, each expressed as an individual function in the code. These functions are appended to the edge detection class, maintaining modularity and readability. By ensuring the edge detection operates with the same input (screenshots) and provides the same output (OpenCV polylines) as the original method, the new implementation can be integrated without requiring large changes to the existing codebase.
The *diagram_tokenization_orchestrator* queries information such as the stroke width and color from the editor model, which could be used in the future to further generalize the edge detection. It also instantiates an object of the edge detection class, passing the workspace path as an argument to extract data like the intersection template. The polylines are stored using a dedicated data storage function, ensuring subsequent steps proceed smoothly without requiering additional integration effort.

4.3 Vertex Detection Integration

The integration of a new vertex detection algorithm into XGEE's codebase required addressing problems such as order-dependent method execution and enabling functions to dynamically share and modify input data at runtime.
Previously, a list of vertices for the current editor, generated by the *diagram_tokenization_orchestrator*, drove the vertex detection process. The method iterated through this list, performing detection cycles for each vertex type, distinguishing between scalable and non-scalable vertices. However, this static approach proved inadequate in the allocations editor due to overlapping vertices, where detecting the top-level vertex first becomes critical. Furthermore, it was not possible for functions to pass intermediate results, such as detected subtasks, to subsequent functions. This was because the input (a preprocessed screenshot) was initialized at the start of the verification process using the *ImageWrapper* class and remained static throughout.
To solve these issues, the new implementation preprocesses the list of editor vertices to ensure vertex detection in the correct hierarchical order and introduces a way for input images to be updated dynamically during runtime.

In the new implementation, the *diagram_tokenization_orchestrator* queries vertex attributes such as the filepath, shape, positioning within other vertices and the parent's filepath from the editor model domain. The vertex's position within the list of editor vertices is determined by the presence of the *isPositioningBody* attribute: vertices with this attribute are classified as subvertices, while those without it are considered top-level vertices. Subvertices need to be detected first, as their overlap with parent vertices would otherwise interfere with the reliable detection of those parent vertices.
Using these attributes, the list of editor vertices is first sorted, prioritizing subtasks. The *diagram_tokenization_orchestrator* then iterates through the sorted list, performing vertex detection for each vertex type. To enable the vertex detection function to alter the input of all subsequent iterations of the function, the creation of the *ImageWrapper* object is shifted inside the vertex detection loop. This enables a new *ImageWrapper* Object to be instantiated for every iteration, using the updated image.
For subvertices, a new function overlays the detected bounding boxes in the target image with the parent vertex's main color (e.g. the gray of device vertices is used to erase the top-level subtasks). The altered image is then used to instantiate the ImageWrapper, ensuring subsequent stages of vertex detection and verification operate on the updated input. This iterative approach enables precise detection of both subvertices and their parent vertices while simplifying the image progressively. After all subvertices are processed, almost all remaining vertices can be detected using the methods described in section 3.2, as the subvertices no longer interfere with the detection process.
An alternative approach for resolving overlapping vertices described in section 3.1 has been implemented for signal container detection in the allocations editor, facilitating the need to process signal containers seperately using a distinct detection class and seperating them from the other vertices using the *filepath* attribute.
Because container vertices are non-scalable, the newly created *ContainerDetector* class inherits from the existing *PortDetector* class (see Figure 4.1). This inheritance allows the *ContainerDetector* to reutilize the majority of the functionality provided by *PortDetector*, while overriding the *detect_vertices* function to implement a specialized detection process tailored to container vertices. Depending on the vertex type, an object of either *ScalableVertexDetector*, *ContainerDetector* or *PortDetector* is instantiated and used to detect the vertices.
The detected vertices are stored using a dedicated data storage function, ensuring subsequent steps proceed smoothly without requiring additional integration effort.

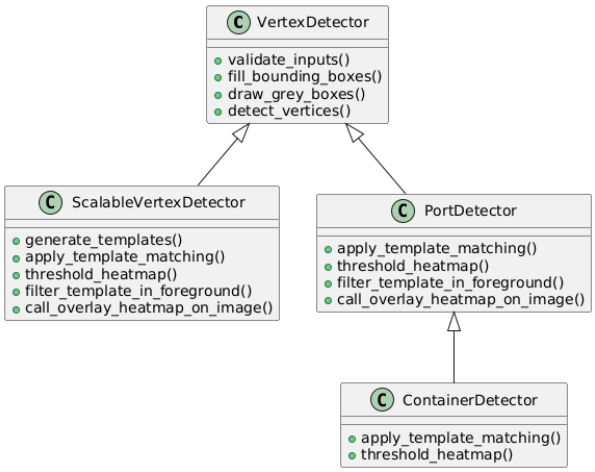


Figure 4.1: Inheritance diagram of the vertex detection classes created using *PlantUML* [19]. The *ContainerDetector* class inherits from the *PortDetector* class, allowing it to reuse the majority of the functionality provided by *PortDetector*.

4.4 Text Recognition Integration

The integration of EasyOCR into the XGEE codebase is achieved by utilizing the EasyOCR API within *diagram_tokenization_orchestrator.py*. The *detect_text* function is updated to employ EasyOCR for text extraction from preprocessed images. The extracted text is then stored using the existing dedicated data storage function, ensuring compatibility with subsequent processing steps. Unlike PyTesseract, EasyOCR includes inconsistent padding around detected text regions, resulting in less precise positional data. This discrepancy caused inconsistencies in positional accuracy and introduced errors in downstream model comparison methods. To ensure compatibility with the already existing interface, the dimensions of the bounding boxes were adjusted to more closely match those of PyTesseract. A positioning margin was implemented as well to reduce the need for bounding boxes to be positioned pixel-perfectly, reducing the amount of interfering error indications.

4.5 Testing and Validation of the Integration

To test the methods, the XGEE editor is used to create models with a known structure, incorporating different test cases in different editor layers. The model is processed by the verification pipeline. If necessary, text, vertex, or edge detections can be turned off to accelerate testing and reduce the workload when evaluating newly implemented methods. The results are compared to the expected output, and any discrepancies are analyzed to identify the cause. To further streamline the testing process, all previously mentioned methods log intermediate images and results using a dedicated Python logging package. This simplifies the process of identifying and resolving new issues.

To classify the performance of all newly implemented methods, 20 unique testcases in each editor model were processed and analyzed to demonstrate the functionality and limitations of the new additions. These evaluations are presented in chapter 5.

5 Results

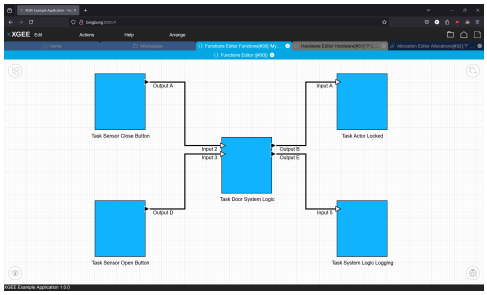
To evaluate the newly implemented functionality, a series of test cases are used. These test cases are created by directly manipulating the model's vector graphics using the *Firefox Developer Tools*. Individual vertices can be manipulated directly within the browser by decoding a *base64*-encoded string that represents the graphic into a readable format that describes the properties of the corresponding *.svg* file. Adjustments to attributes such as *fill* followed by re-encoding the data into *base64* allows changing the appearance of the visualization. Similarly, vertex positions can be modified, or vertices can be removed entirely, by adjusting the relevant values within the developer tools.
All test cases are derived from the models shown in Figure 5.1 and are simulated with minimal deviations to ensure accuracy.

5.1 Test Cases

Because the Error indications differ based on the step of the pipeline the error was found in, the testcases are divided into three categories:

- Testcases where the error can be indicated textually as well as visually highlighted inside the editor (see Table 5.1).
- Testcases where the error can only be indicated via a textual error indication (see Table 5.2).
- Testcases without any simulated errors to illustrate the pipeline's capability to deal with complex models (see Table 5.3).

Table 5.1 shows the results of testcases where the error can be identified and visually highlighted inside the XGEE Editor, as well as indicated via the textual log. The first column in Table 5.1 provides a brief description of the simulated error and the used error model. The second column provides an image of the relevant portion of the screenshot with the error highlighted. If the error type is found to be *Untokenized Pixels*, the highlighting color is purple, otherwise, the color is red. The third column provides the textual error indication alongside the step of the verification pipeline, where the error was detected. In some cases, for example when a device with multiple IOs is removed or too small to be detected, many error indications of the same type are generated. In this case, only a few of the error indications are shown in the table and further errors of the same type are indicated using " ⋮ ".



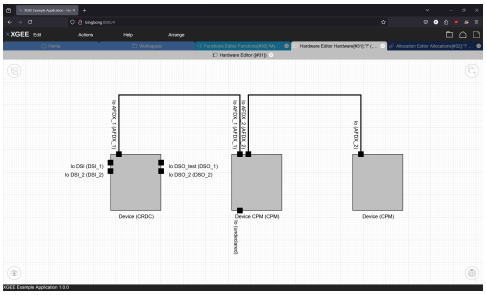
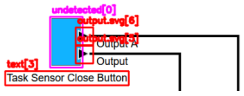
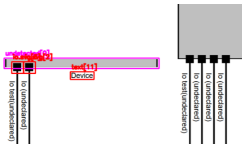
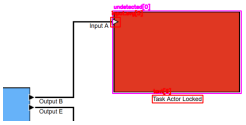
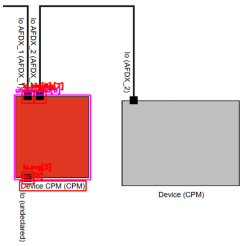
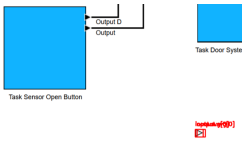


Figure 5.1: Unaltered models as a basis for testcase generation.
Left: Functions Layer, Right: Hardware Layer

The testcases enumerated in Table 5.2 only result in textual error indications. This is because the tested errors can not be detected during the tokenization or syntax steps of the verification pipeline, meaning that visually, the used tokens and their syntax are correct. The errors are only discovered during the comparison or instantiation steps, resulting in only a textual error indication. The first column of the table provides a zoomed-in view of a model containing the error. The second column provides a brief description of the testcase and the indicated textual errors.

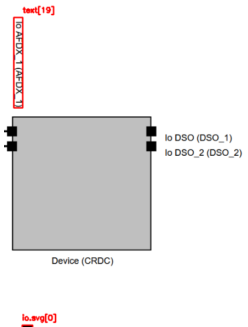
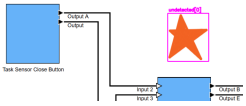
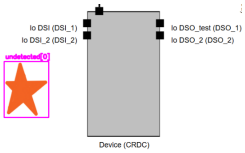
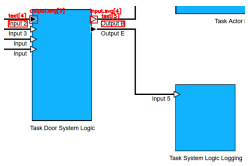
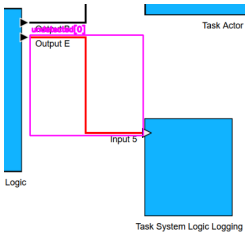
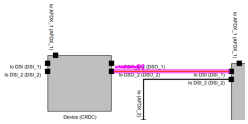
Table 5.3 contains testcases without any simulated errors to illustrate the pipelines capability to deal with complex models including diverse vertices and intersecting connections. The first columnn provides a brief description, the second and third column provide the visualization within the functions layer and within the hardware layer of the XGEE editor.
While the tokenization is functional for three of XGEEs model layers, the allocations layer has not yet been implemented fully into the verification pipeline, meaning that there are currently no meaningful error indications when trying to verify a model in the allocations layer. Hence, the testcases are only shown for the functions and hardware layers.
A seperate testcase showcasing the capability of the tokenization pipeline of allocation models can be seen in Figure 3.19. All found edges, bounding boxes and their respective token names are overlayed in green on the screenshot of the used model.

Table 5.1: Results for test cases with textual and visual error indication.

Test Case	Excerpt Visual Error Indication	Textual Error Indication
Functions Layer - Task too Small		**Tokenization**: untokenized pixels. **Syntax**: ['text', 3] is missing a parent element. **Syntax**: ['output.svg', 5] is missing a parent element. **Syntax**: ['output.svg', 6] is missing a parent element. **Instantiation**: output.svg[5] is not instantiated, association cannot connect. **Comparison**: Task Sensor Close Button missing in recognized model. **Comparison**: Signal Door System Logic: B → Actor Locked: A missing in recognized model. ⋮
Hardware Layer - Device too Small		**Tokenization**: untokenized pixels. **Syntax**: ['text', 11] is missing a parent element. **Syntax**: ['io.svg', 2] is missing a parent element. **Syntax**: ['io.svg', 3] is missing a parent element.
Functions Layer - Task Wrong Color		**Tokenization**: untokenized pixels. **Syntax**: ['text', 6] is missing a parent element. **Syntax**: ['input.svg', 5] is missing a parent element. **Instantiation**: input.svg[5] not instantiated, association cannot connect. **Comparison**: Signal Door System Logic: B → Actor Locked: A missing in recognized model. **Comparison**: Signal Door System Logic: E → System Logic Logging: 5 missing in recognized model. ⋮
Hardware Layer - Device Wrong Color		**Tokenization**: untokenized pixels. **Syntax**: ['text', 10] is missing a parent element. **Syntax**: ['io.svg', 0] is missing a parent element. **Syntax**: ['io.svg', 1] is missing a parent element. ⋮
Functions Layer - Free Floating Input		**Syntax**: ['input.svg', 0] is missing a parent element. **Syntax**: ['output.svg', 1] is missing a parent element.

Continued on next page

Table 5.1 continued from previous page

Test Case	Excerpt Visual Error Indication	Textual Error Indication
Hardware Layer - Free Floating IO		**Syntax**: ['text', 19] is missing a parent element. **Syntax**: ['io.svg', 0] is missing a parent element.
Functions Layer - Free Floating Star		**Tokenization**: untokenized pixels.
Hardware Layer - Free Floating Star		**Tokenization**: untokenized pixels.
Functions Layer - Input Instead of Output		**Syntax**: ['text', 4] is missing a parent element. **Syntax**: ['text', 4] is missing a parent element. **Syntax**: ['input.svg', 4] is missing a parent element. **Syntax**: ['output.svg', 3] is missing a parent element. **Instantiation**: Failed to instantiate association. Endpoint input.svg. **Instantiation**: Failed to instantiate association. Endpoint output.svg. **Comparison**: Signal Door System Logic: B → Actor Locked: 1 missing. ⋮
Functions Layer - Signal Wrong Color		**Tokenization**: untokenized pixels. **Syntax**: ['text', 17] is missing a parent element. **Comparison**: Number of Signals in the original model (6) does not match (5). **Comparison**: Signal Door System Logic: E → System Logic Logging: 5 missing
Hardware Layer - Signal Wrong Color		**Tokenization**: untokenized pixels.

Continued on next page

Table 5.1 continued from previous page

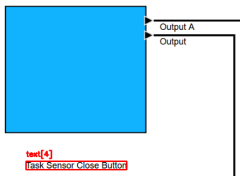
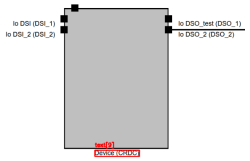
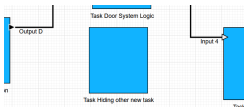
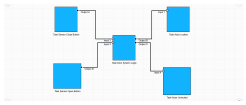
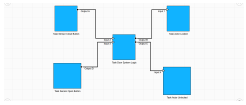
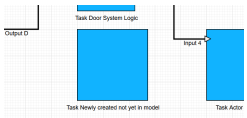
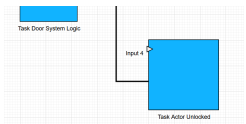
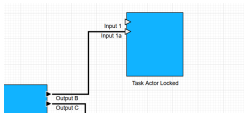
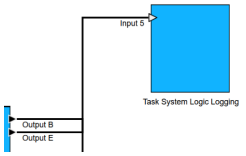
Test Case	Excerpt Visual Error Indication	Textual Error Indication
Functions Layer - Text too Far		**Syntax**: ['text', 4] is missing a parent element. **Comparison**: Task "Sensor Close Button" missing in recognized model. **Comparison**: Signal Sensor Close Button: A → Door System
Hardware Layer - Text too Near		**Syntax**:['text', 9] is missing a parent element.

Table 5.2: Results for test cases with textual error indication.

Test Case	Excerpt Modified Screenshot	Textual Error Indication
New Task Hiding Other New Task		**Comparison**: Task Hidden missing
Task Out of Window		**Comparison**: Task Far Away missing
Task Deleted in Editor, but Still in Model		**Comparison**: Task New Task Already Deleted Missing
Task Created, but Not Yet in Model		**Comparison**: Task Newly Created Not Yet in Model Missing in Original Model
Signal Not Connected to Input		**Comparison**: Signal Door System Logic:C → Actor Unlocked:4 Missing
Signal Connected to Wrong Port		**Comparison**: Signal Door System Logic: B → Actor Locked: A Missing **Comparison**: Signal Door System Logic: E → System Logic Logging: 5 Missing
Overlapping Intersections		**Comparison**: Sensor Open Button:D → Door System Logic:3 Missing **Comparison**: Task Sensor Open Button Missing

Continued on next page

Table 5.2 continued from previous page

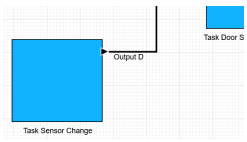
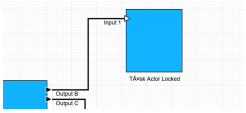
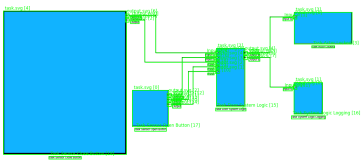
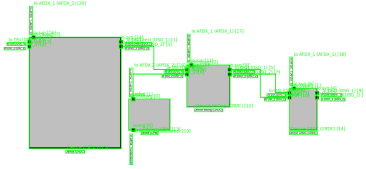
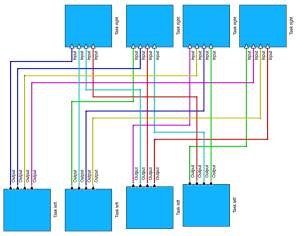
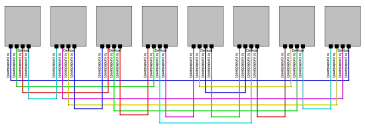
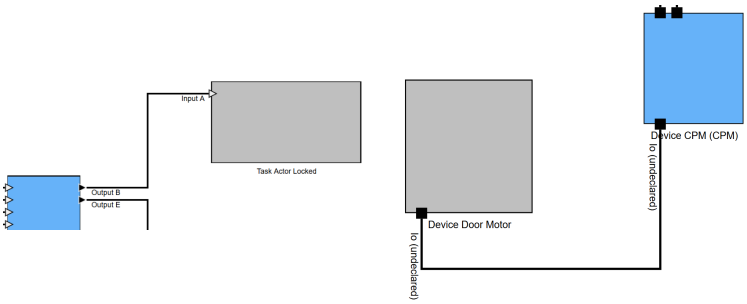
Test Case	Excerpt Modified Screenshot	Textual Error Indication
Changed Task Name, but Not Yet in Model		**Comparison**: Sensor Open Button:D → Door System Logic:3 Missing **Comparison**: Task Sensor Open Button Missing
Encoding Error in Task Name		**Comparison**: Task Actor Locked Missing **Comparison**: Signal Door System Logic:B → Actor Locked:1 Missing

Table 5.3: Functional Test Cases Demonstrating Correct System Behavior

Test case description	functions layer visualization	hardware layer visualization
Processing of vertices with varying sizes and aspect ratios		
Recognition of complex signal intersections in close proximity		
Function / device has the wrong color - but is still detected by the template matching method.		

6 Discussion

This chapter highlights the implemented methods' capability to handle a variety of testcases, as well as current limitations and challenges of the implemented methods, focusing on their impact on error detection, visualization, and overall system efficiency.

6.1 Discussion of Test Cases

As seen in chapter 5, all 20 unique testcases are correctly identified and indicated to the user and, as seen in Table 5.3, no error smells are reported if none are simulated. (The concept of *error smells* describes patterns or indicators that suggest the presence of errors in the system.) In the *free floating star* test case, the star is detected and highlighted as untokenized during the tokenization step at the beginning of the pipeline. In the *input instead of output* and *free floating input / IO* testcases, all vertices are known, but they are not correctly positioned, resulting in an error smell indication during the syntax step of the pipeline. Similar indications are generated in the *text too far* and *text too near* test cases, because they too are correctly identified during tokenization but have a slightly shifted position, resulting in syntax error indications.

In *device wrong color*, *task wrong color* and *signal wrong color*, the color of one of the vertices or edges is changed such, that it is no longer being detected in the tokenization step. This is because OpenCV's *templatematching* method does not consider color when comparing the template image with the underlying pixels. Similarly, the *match2D* method only considers grayscale pixels when detecting edges. When the color changes, the grayscale values of the pixels change as well. Most of the time, this results in untokenized pixels, as well as a number of additional error smells, because for example, the associated IO ports of the changed device are now seemingly floating in space, resulting in wrong syntax. The same problem arises in the simulated testcases *task / device too small*.

Further problems arise when instantiating and comparing the new model without the changed device. These additional error smells are reported, making it harder for the user to identify the original problem cause.
Furthermore, as illustrated in Table 5.3 *row 3*, an unexpectedly changed color is only indicated to the user, if the intensity of the original and altered color is significantly different. This is because OpenCV's *matchtemplate* function compares the pixels intensities, effectively ignoring colors. This could be improved by running the template matching algorithm seperately for each color channel and then comparing the results or by incorporating a different method more suitable for differentiating color hues.

6.2 Limitations of the Current Implementation

The current implementation is, in large parts, model driven, meaning it can recognize the current editor model and query it for information, such as a list of the used token types and the filepath of their respective *.svg* files. This information can then be used to dynamically change the verification pipeline, for example by choosing the correct templates for the current visualization. When the visualized model, which is the one seen by the user, changes unexpectedly, the editor model still provides the necessary information to run the verification pipeline.
However, a current limitation is the amount of information that is queried and utilized. For instance, the current implementation does not query the current model layer, making it difficult to differentiate between *function-*, *hardware-*, and *allocation* layers. This can lead to less

efficient tokenization. For example, during text recognition, the image is searched for rotated text, regardless of whether the current model contains any.
Another limitation lies in the template matching method. As stated in chapter 3, the current implementation applies each template to the image, which works well if the used templates are stored within the model, enabling the system to query them and dynamically choose the correct templates for each model layer. However, for edge and intersection detection, no such convenient template files are stored. Instead, the templates are generated with a fixed size, making it harder to adapt the system to changes in edge width, color, or intersection visualizations.
Another limitation stems from the visualization and the edge detection pipeline's difficulty in identifying the edges connecting subtasks within the allocations editor (see Figure 6.1). This is because these edges are too thin and lack sufficient contrast for accurate detection. Implementing a more robust algorithm or designing a visualization that is easier to detect for computer vision algorithms could enhance the detection process, potentially allowing it to be integrated into the edge detection pipeline.

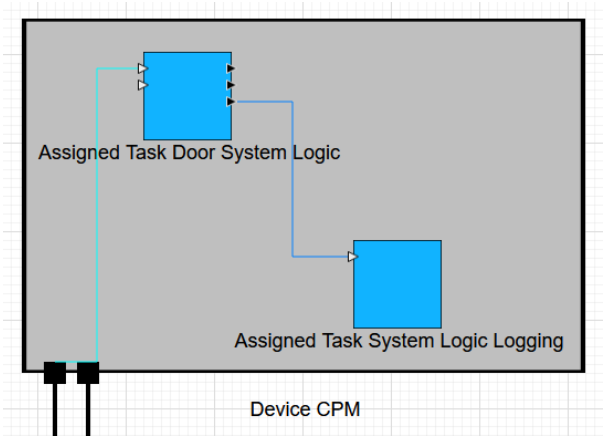


Figure 6.1: Example of the thin edges in the allocations editor, which are hard to detect using the current edge detection method.

Another limiting factor is the lack of error handling mechanisms. If an error occurs during the verification, it can disrupt or stop the entire workflow. Furthermore, a single error smell can result in many error indications, as discussed in chapter 6, hampering the user's ability to quickly find the root cause of the error indications.

All of these limitations could be addressed by enhancing the current implementation with additional features, such as querying the model for more information, dynamically generating templates, or implementing more robust error handling mechanisms.

7 Outlook

This chapter explores potential improvements and future directions for the methods and tools introduced in this thesis, with a focus on insights gained from the analysis of test cases. While the proposed edge, vertex, and text detection algorithms perform reliably in most scenarios, certain limitations become evident when applied to complex or ambiguous test cases.

7.1 Edge Detection Further Improvements

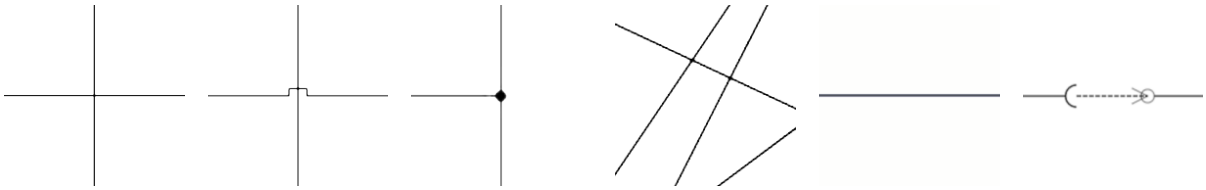


Figure 7.1: Example of different intersection and edge types, which would require different handling in the edge detection pipeline.

The method introduced in this thesis 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 models, provided the models are formatted correctly (the zoom level has to be set to 100% across the entire operating system for the edge detection to work correctly). In cases where multiple edges overlap or many intersections are within a few pixels of each other, an error smell is reported, as the method, as well as human operators, can not reliably interpret the image. This enforces an unambiguous model layout, which is achieved by either an automatic arrangement algorithm or the user.

The current intersection detection relies on a very specific visualization, where edges run in straight lines and intersecting edges simply cross. There are, however, many possible ways to visualize edges and intersections (see Figure 7.1). This poses a limitation, as the current method cannot adapt if the edge and intersection visualization changes, for instance, to better display the intersection of multiple edges. A potential solution to this problem could involve extracting the intersection graphic from the screenshot by predicting the intersection's position based on the detected edges. This approach would enable the method to adapt to different intersection visualizations automatically, as long as the edges remain detectable. Another potential solution would be to include information about the possible intersection types within the editor model, enabling the method to generate a template image.

A notable problem arises when vertices with surrounding black edges, such as devices or functions, are scaled sufficiently, making their edges resemble signal-carrying edges (see Table 5.3 first testcase, where the black borders around the larger vertices appear wider than those around smaller vertices). This complicates their differentiation. In these cases, 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 more thoroughly, 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 possible solution would be to implement an enhanced scaling method into XGEE, eliminating

the problem of vertex borders scaling together with vertex bodies, thus resembeling edges.

Another issue can arise if the edges are configured by the user in an unexpected way. For instance, two overlapping vertices, such as in Figure 7.2, can not be properly interpreted by human users, was well as computer vision algorithms. The text labels can not be read, the vertex borders are not clear and the edges overlap, making it impossible to clearly read the diagram structure. Future improvements to the XGEE editor, such as a more advanced automatic arrangement algorithm, could reduce or eliminate the risk of ambiguous user input that would result in such cases.
In less problematic cases, for example an intersection hidden behind a single vertex, a future method could search for edges along the sides of detected devices and functions and attempt to connect them in a meaningful way. However, this approach might fail if multiple edges run behind a single device or function, which would also make the visualization difficult for a human user to interpret.

Another issue arises from edges connecting subtasks within the allocations editor. They often overlap with text, are too thin, and have too little contrast to be detected accurately. In a future version of XGEE's visualization, enhancing the readability of subtask edges would enable a computer vision algorithm to detect them reliably, enabeling useful user feedback and thus improving the verification tool.

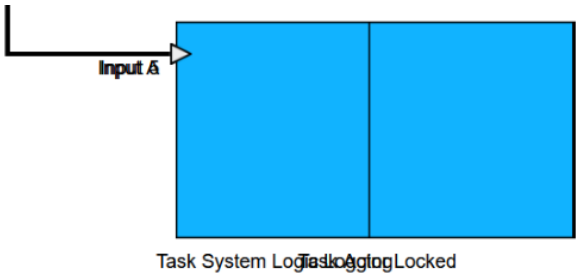


Figure 7.2: Overlapping edges, vertices and labels in the functions editor, which lead to ambiguous edge, text, and vertex detections.

7.2 Vertex Detection Further Improvements

The proposed method can reliably detect almost all vertices across three of XGEE's model layers, regardless of their dimensions or placement. It can effectively distinguish between subvertices and main vertices by querying the model and applying the appropriate template matching algorithm based on the properties of each template. Additionally, the current editor model is queried to identify the set of utilized vertices, which are subsequently detected. This adaptability enhances both the efficiency and reliability of the method, enabling it to handle changes in the model.
As described in chapter 3, the current method identifies each token type within the allocation layer by following a structured detection sequence. First, subtasks are identified and subsequently erased from the screenshot to prevent interference with the detection of underlying vertices.
Currently, this approach is only used to enable the detection of devices in the allocations layer. In the future, extending this approach across the entire tokenization pipeline could significantly enhance vertex detection, ensuring that no vertices are obscured, no vertex is detected multiple times, and similar vertices, edges, or text are not mistakenly identified as one another.
To implement this improvement, the following steps could be taken:

- Process all vertices, edges and labels in hierarchical top-down order, starting with subvertices and ending with the main vertices, followed by labels and edges.
- Query the parent vertices for their main color and use it to erase all found vertices.
- Dynamically update the input image after each iteration and pass it to the next step in the tokenization pipeline.
- Correctly handle cases where subvertices, such as IO ports, only partially overlap with their parent vertices.

This approach would systematically simplify the screenshot with each step of the tokenization pipeline, effectively only leaving the untokenized pixels on a white background in the end. After the vertex detection step, the remaining pixels would be passed to the text detection algorithm, which would then identify any remaining text. Finally, the edge detection algorithm would process the remaining pixels, detecting any edges without the possibility of interference from other vertices. This approach would, however, require a way to deal with overlapping text labels, as they would most likely be partially removed in previous steps, making it impossible to properly recognize them.

Another possible improvement could be made by enabeling the current method to detect subvertices of subtasks (very small inputs and outputs, as seen in Figure 6.1), which are currently excluded from detection due to their small size. These *subsubvertices* are challenging to differentiate from noise, character fragments, or edge segments using OpenCV's built in methods. Employing image preprocessing techniques or refining or exchanging the template matching method could improve the detection of these smaller elements and thus enable a more precise visualization verification.

7.3 Text Detection Further Improvements

Currently, the text recognition system can detect almost any text present within XGEE. Regardless of the current editor, the screenshot is rotated and analyzed to identify any rotated text, which contributes to text detection being the most time consuming component of the visualization verification process.

Optimizing the performance of the text detection algorithm would significantly reduce the overall processing time. This could be achieved by identifying the specific editor type to allow the system to selectively search for rotated characters only when they are expected to be present in the image. Additionally, running the visualization verification on a GPU, parallelizing the text detection to run simultaneous to the edge- and vertex detection or using a more traditional text detection algorithm which does not require as much computational power as EasyOCR would enable the system to process larger models more quickly and efficiently, addressing the current bottleneck in the pipeline.
Currently, the average runtime of the verification pipeline on an Intel Core Ultra 9 185H CPU is 54 seconds, with 76% of runtime spent on character recognition.
A new OCR model would also potentially allow for a more precise way to filter out falsely read characters instead of removing every recognized word shorter than three letters.

Another potential improvement could be made regarding the text recognition's accuracy. Currently, as seen in Figure 7.3, text labels are sometimes not correctly detected, resulting in false

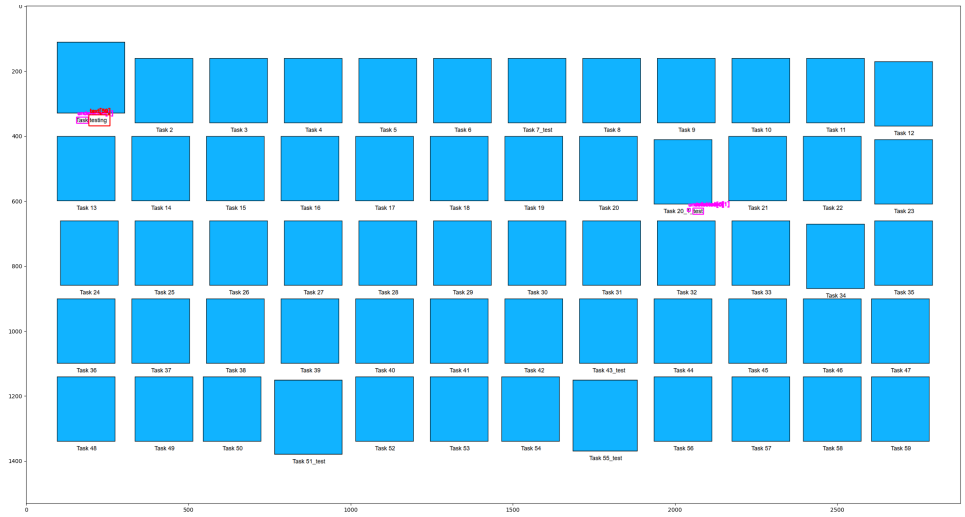


Figure 7.3: Example debugging image with many tasks. The current text recognition is not accurate enough to reliably detect all text labels in all images, making it necessary for the user to manually check the verification results in some cases (misidentified labels are marked as in section 5.1).

positive error indications, that would have to be checked manually by the user. The method is limited by the quality of the input image and the EasyOCR library. This is because most OCR engines are trained on books, where text is orderly structured and has a predictable orientation. In contrast, the text in XGEE models is often spread out unevenly, rotated, distorted, or partially occluded, making it difficult for the OCR engine to recognize. Additionally, individual numbers are hard for EasyOCR to detect reliably. In these cases, the user has to check the verification results manually.
Training a custom OCR model on a dataset of XGEE text images could improve the recognition accuracy, as the model would be specifically tailored to the unique characteristics of XGEE text. In future implementations, this could be achieved by automatically generating the training data alongside the correct text by directly querying the model.
A simpler way to solve the problem of wrongly detected characters could be improved error handeling, to indicate these types of errors to the user in a simple and clean way. This would enable the user to quickly identify and correct any false errors, reducing the time spent on verification.

7.4 Expanding to other Domains or Applications

As described in chapter 5, the current implementation is optimized to work model driven in the functions- and hardware layer. Developing the implementation further to enable verification of the allocations layer would be a logical next step, fully enabeling the verification tool to work with three of XGEE's editors, dynamically changing with the model. However, because of the higher complexity of the allocations layer, this would require restructuring the current implementation to consider the order of detection and the hierarchical structure of the model in all steps of the verification pipeline.

One of the goals of this thesis was to generalize the used tokenization algorithms to work in the functions- and hardware editor of XGEE. Expanding upon this idea, a future verification pipeline could be able to understand a wide range of different editors, potentially enabeling the verification of editors completely seperated from the verification program. This could be

achieved by implementing a more general tokenization pipeline, which could be configured to work with any editor, given the correct templates and cofiguration files.
Enabeling the verification to work with editor models like Simulink and other widely used modeling tools would make the verification tool more versatile and useful to a wide audience, potentially resulting in an increase in demand for automatic visualization verification tools.

Bibliography

[1] A. Waldvogel, "Towards qualifiable graphical editing of complex domain-specific models in safety-critical avionics," *MODELS: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2022. DOI: 10.1145/3550356.3558507.

[2] A. Waldvogel and B. Annighoefer, "Model-based block diagram recognition for model visualization verification," *MODELS Companion: 27th International Conference on Model Driven Engineering Languages and Systems*, 2024. DOI: 10.1145/3652620.3687825.

[3] "The unified modeling language," Accessed: Feb. 2, 2025. [Online]. Available: https://www.uml-diagrams.org/.

[4] S. Mani, M. A. Haddad, D. Constantini, W. Douhard, Q. Li, and L. Poirier, "Automatic digitization of engineering diagrams using deep learning and graph search," *Conference on Computer Vision and Pattern Recognition Workshops*, 2020. DOI: 10.1109/CVPRW50498.2020.00096.

[5] X. Wei, S. L. Phung, A. Bouzerdoum, and A. Bermak, "Invariant image recognition under projective deformations: An image normalization approach," *Visual Communications and Image Processing*, 2015. DOI: 10.1109/VCIP.2015.7457793.

[6] B. Schaefer, M. Keuper, and H. Stuckenschmidt, "Arrow r-cnn for handwritten diagram recognition," *International Journal on Document Analysis and Recognition*, 2021. DOI: 10.1007/s10032-020-00361-1.

[7] Z. Aston, L. Z. C, L. Mu, and S. A. J, *Dive into Deep Learning*. Cambridge University Press, 2023, https://D2L.ai.

[8] J. Fang, Z. Feng, and B. Cai, "Drawnnet: Offline hand-drawn diagram recognition based on keypoint prediction of aggregating geometric characteristics," *Entropy*, 2022. DOI: 10.3390/e24030425.

[9] L. Yang *et al.*, "Practical single-line diagram recognition based on digital image processing and deep vision models," *Expert Systems with Applications*, 2024. DOI: 10.1016/j.eswa.2023.122389.

[10] "Making your own linear filters," Accessed: Nov. 18, 2024. [Online]. Available: https://docs.opencv.org/3.4/d4/dbd/tutorial_filter_2d.html.

[11] S. Suzuki *et al.*, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, 1985. DOI: 10.1016/0734-189X(85)90016-7.

[12] 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*, 1973. DOI: 10.1002/9780470669488.

[13] "Opencv documentation matchtemplate," Accessed: Nov. 28, 2024. [Online]. Available: https://docs.opencv.org/3.4/df/dfb/group__imgproc__object.html#ga586ebfb0a7fb604b35a23d85391329be.

[14] "Matchtemplate comparison methods," Accessed: Nov. 28, 2024. [Online]. Available: https://docs.opencv.org/3.4/df/dfb/group__imgproc__object.html#ga3a7850640f1fe1f58fe91a2d7583695d.

[15] "Pytesseract 0.3.13," Accessed: Jan. 24, 2024. [Online]. Available: https://pypi.org/project/pytesseract/.

[16] rkcosmos. "Easyocr 1.7.2," Accessed: Dec. 3, 2024. [Online]. Available: https://pypi.org/project/easyocr/.

[17] techmindee. "Python-doctr 0.10.0," Accessed: Dec. 3, 2024. [Online]. Available: https://pypi.org/project/python-doctr/.

[18] faustomorales. "Keras-ocr 0.9.3," Accessed: Dec. 3, 2024. [Online]. Available: https://pypi.org/project/keras-ocr/.

[19] "Uml diagrams with plantuml," Accessed: Feb. 3, 2025. [Online]. Available: https://www.plantuml.com/plantuml/uml/.

[20] "Template matching," Accessed: Nov. 18, 2024. [Online]. Available: https://docs.opencv.org/3.4/d4/dc6/tutorial_py_template_matching.html.

[21] "Opencv documentation findcontours," Accessed: Nov. 28, 2024. [Online]. Available: https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#ga17ed9f5d79ae97bd4c7cf18403e1689a.

[22] S. T. Karri. "Hough transform," Accessed: Jan. 23, 2025. [Online]. Available: https://medium.com/@st1739/hough-transform-287b2dac0c70.

[23] "Oaam - open avionics architecture model," Accessed: Feb. 2, 2025. [Online]. Available: https://github.com/ComplexAvionicsSystems/OAAM.

[24] "Make simulink line crossings more distinct," Accessed: Feb. 4, 2025. [Online]. Available: https://blogs.mathworks.com/pick/2012/10/05/make-simulink-line-crossing-more-distinct/.

[25] "Model and validate a system," Accessed: Feb. 4, 2025. [Online]. Available: https://www.mathworks.com/help/simulink/gs/model-and-validate-a-system.html.

[26] R. Szeliski, *Texts in Computer Science - Computer Vision - Algorithms and Applications Second Edition*. Springer, 2022, ISBN: 978-3-030-34374-3.

[27] S. Kelly and J. P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, 2007, ISBN: 978-0-470-03666-2.

[28] B. Annighoefer and A. Waldvogel, *Domain-specific modeling and analysis*, Lecture at the University of Stuttgart, 2024.