Automated Testing of Interaction-Requiring Devices with a Robotic Arm and Object Detection

Levente Daroczi*, Beniamin Demeter*, Gyöngyi Katona†, Dénes Mihály†, Csaba Sulyok*, Arnold Szász†
*Faculty of Mathematics and Computer Science, Babeş–Bolyai University
RO-400084 Cluj-Napoca, Romania
E-mail: {levente.daroczi, beniamin.czompa}@stud.ubbcluj.ro, csaba.sulyok@ubbcluj.ro
†Codespring
RO-400458 Cluj-Napoca, Romania
E-mail: {katona.gyongyi, mihaly.denes, szasz.arnold}@codespring.ro

***Abstract*—Testing user interfaces is fundamental for ensuring reliability and proper functionality. However, for many software systems, there is no framework available that would support automated testing, especially when the system requires manual input controls such as physical buttons. Currently, testing of these devices is largely done manually, which is extremely time-consuming, requires a high degree of attention, and can increase the potential for errors.**

The project aims to create an automated testing system that uses a robotic arm and object detection technology. The system is capable of testing devices with physical buttons and mechanical controls, ensuring precise and repeatable interactions, as well as validating device feedback. This reduces the need for manual intervention while significantly increasing the efficiency of the testing process.

The solution consists of a web application and a backend server. The web application provides the user interface that allows for creating tests and viewing results, while the backend server is responsible for executing the tests, controlling the robotic arm, and performing object detection processes.

***Index Terms*—object detection, robotic arm, automation, testing, AI**

I. Introduction

In the domain of software development, integration testing of a finished product is at least as important as the separate verification of individual components. While the latter ensures the correct operation of the building blocks, the former guarantees their proper cooperation [1]. Testing, however, becomes significantly more complex when the software is not running on a general-purpose computer but is instead designed for dedicated hardware [2]. In such cases, standard testing infrastructure is often lacking, especially for microcontroller-based devices [3]. Without such support, it is difficult or even impossible to issue automatic test instructions or to retrieve their results from the target hardware. The challenge is further increased if the device is equipped with physical input controls that require human interaction, such as push buttons, touchscreens, or rotary switches. Testing these using conventional tools can be particularly cumbersome, especially once the product has completed its final manufacturing phase [4], [5].

The current paper proposes a solution for automated testing of such unconventional systems, relying on the following tools: a robotic arm, a camera, and advanced object detection software. The arm simulates human interaction by handling physical inputs such as button presses, touchscreen navigation, or turning rotary switches. The recognition unit assists in controlling the robotic arm and enables result verification. Its primary task is to identify the components of the device under test through the camera feed.

Unlike conventional testing methods, the use of a robotic arm is essential for devices equipped with physical input interfaces that require human interaction. While various testing frameworks exist–such as *Appium*[1], which communicates with Android devices using the *Android Debug Bridge (ADB)*[2] for Android devices or software specialized in UI/UX testing-these solutions are typically limited to a specific platform or input method. There is a lack of a universally applicable system capable of handling devices with diverse input interfaces, such as touchscreens, physical buttons, or rotary switches.

With an accurate description of components of the test device and deep learning-based recognition models, complete multi-step tests can be executed without human intervention. Introducing an automated system offers several advantages over traditional manual testing. It eliminates human errors and significantly frees up time for test engineers [4].

Currently, there are few solutions on the market capable of software testing for devices requiring interaction by combining robotic arm and object detection technology. Among the best-known similar systems are *MATT*[3], developed by *Adapta Robotics*, and *QUACO Pro*[4] from the portfolio of *Sastra Robotics*. These solutions use proprietary devices that have been specifically designed for this purpose.

II. Interface Element Recognition in Images

The current project aims to allow navigation on/interaction with any user interface, including touchscreens, physical buttons, knobs and other actuators. The current section focuses on the technology choices enabling this.

[1]Source: https://appium.io/docs/en/latest/
[2]Source: https://developer.android.com/tools/adb
[3]Source: https://www.adaptarobotics.com/matt/
[4]Source: https://sastrarobotics.com/products/quaco-pro/

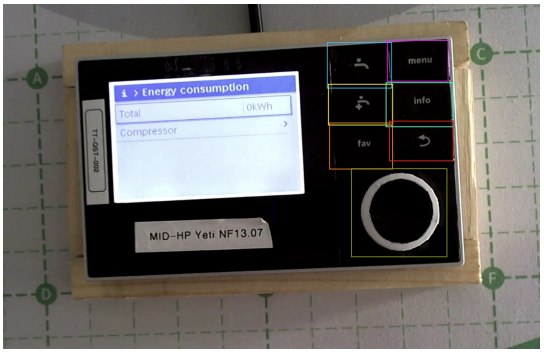

Fig. 1. Buttons of the controller used for managing heating systems, with labelled elements

Object detection algorithms are capable of recognizing different shapes, icons, and even shorter text. They can detect specific visual patterns and structures, which help identify buttons, surfaces, and other elements [6]. The detection employed in this project is based on deep learning, and enables the identification and localization of various objects in images or videos. The goal of their usage is to detect UI widgets on screens, read label texts and identify interactive elements [7].

Among the various technologies, the YOLO (You Only Look Once) algorithm family [8], [9] is chosen, as it is capable of identifying multiple objects in a single image. This is particularly important on a user interface, where real-time and accurate detection plays a key role. YOLO is a single-stage detector that has undergone numerous developments over the years, continuously integrating the latest methods into its architecture. The current project uses YOLOv8 as it was the newest and most advanced version at the time of development.

YOLO recognizes objects and their positions with "a single glance", that is, with a single image processing, hence its name. It uses a convolutional neural network (CNN) to predict the bounding boxes of different objects in an image, as well as their associated probabilities. CNNs are very effective at processing visual data, as features can efficiently pass from initial convolutional layers to later ones.

The first step in model training and object detection is creating a well-structured and properly labelled dataset. In order for the models to accurately and reliably recognize the desired objects under all conditions, the dataset is created in a varied environment. Images are captured under different light conditions, continuously changing light sources, and the device to be tested is photographed from multiple angles and in different positions. Additionally, the camera position is regularly modified so that the model would be able to handle perspective differences (see Fig. 1).

After the training process, the result contains images and files showing various statistics that help evaluate the performance of the model and the best-performing model.

The project contains a dedicated recognition layer that is responsible for identifying different objects based on given parameters. In order for this layer not to be specific to a particular test device, it has been designed so that no modifications to this layer are necessary when introducing additional devices.

Fig. 2. The robotic arm and testing mat recorded by a fixed camera

III. Robotic arm

The application utilizes the *Magician Lite* robotic arm developed by *Dobot Robotics* (see Fig. 2) for controlling the test device. This is a general-purpose device designed for educational purposes, which can be controlled via hardware, software, or Python programs [10].

The arm is capable of executing instructions with a repeatability of 0.2 mm [10], providing more than adequate precision for the use case of this project. Multiple types of end effectors can be attached to the end of the arm, such as a pen holder, a soft gripper, or a rotatable suction cup capable of vacuum-based gripping of small objects (see Table I).

From a hardware perspective, out of the three available end effectors, only two are actively used, as the gripper unit does not prove useful in this application environment. The pen holder unit, combined with a touch-sensitive stylus, is suitable for controlling touchscreen devices such as a smartphone. The suction cup unit can be effectively used to operate devices equipped with push buttons or rotary knobs.

The design of the system enables the automatic testing of devices with varying types, sizes, and input interfaces. However, integration into the application requires the

TABLE I
DOBOT Magician Lite end effectors [10]

End Effectors	
Pen holder	Pen diameter: 8-12 mm
Suction cup	Built-in air pump drive, operates under negative pressure, with suction cup diameters of 10 mm and 20 mm
Soft gripper	Built-in air pump drive, operates under both positive and negative pressure, maximum opening and closing distance: 50 mm

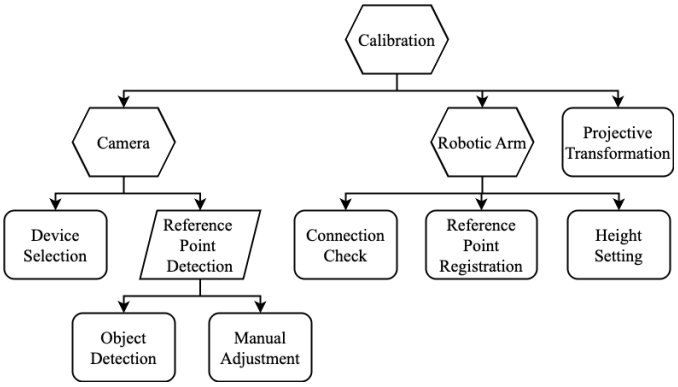

Fig. 3. Behaviour tree of the calibration process

following conditions to be met: (1) the surfaces of the device intended for control or observation must be located within the working area of the arm; (2) these components must be placed on the upper side of the device; (3) the input interfaces must be physically accessible and operable by the arm. Supported components include touchscreens, LCD or other displays, push buttons, rotary knobs, and other physical control elements.

A. Robotic arm and camera

The robotic arm comes with a special mat that determines the position of the arm base and defines its workspace (see Fig. 2). While these markings are not strictly necessary for the operation of the robotic arm, the six reference points (*A–F*) on the mat play a key role in the system. On one hand, they visually designate the workspace defined by the application, but their most important function is to support calibration.

To align the camera image with the other device coordinate system, at least four points with known coordinates are needed both in the camera image and within the workspace. The reference points placed on the mat serve precisely this purpose: with human assistance, the arm can read their spatial coordinates, while they also appear as recognizable objects in the camera image. This ensures that visual perception and arm movement are precisely synchronized.

The camera is an essential component of the application, as it enables the recognition of test device components and plays a crucial role during the calibration of the robotic arm. During a test run, whenever component detection is required, the control unit retrieves the current frame from the camera and runs the YOLO model on it. For calibration, the reference points on the mat are also identified using a pre-trained YOLOv8 model executed on the camera image.

The primary goal in programming the device is to enable the simplest possible usage during test execution. For this purpose, an abstraction layer covers the methods of the aforementioned library.

B. Calibration

Precise *calibration* of the arm is essential for running tests. Early tests demonstrate that the pixel coordinates of components detected by object detection algorithms and the arm's own coordinate system are not compatible. This is resolved by applying a *projective transformation*, which enables the mapping between the two planes if at least 4 reference points are precisely known for both. The more available points, the more accurate the fit [11]. The coordinates of the six reference points on the special mat can be retrieved from the camera image via the camera module, using the previously trained YOLOv8 model. In the coordinate system of the robotic arm, these must be registered manually, which requires human intervention. To ensure easy development and modularity, calibration is outlined and executed using a behaviour tree (see Fig. 3).

Device Selection is an atomic element that waits for the input of the user. Once the camera is selected, it initiates camera initialization. In case of error, this information propagates to the top level, and calibration becomes invalid. This reaction is similar in all cases, except for components capable of error handling.

Reference Point Detection is a selector-type component that also executes its descendants from left to right but stops at the first successful execution and passes the result up. According to the figure, detection is first attempted with object detection, and if unsuccessful, manual adjustment is required. If detection is successful, no further intervention is needed.

During execution of the atomic *Object Detection* element, the previously mentioned YOLOv8 model for reference point detection is run. To reduce errors, the model is retried up to five times in case of unsuccessful detection. If no satisfactory result is achieved after this, the component signals an error. The behaviour tree library allows information sharing via a key-value storage system called the *blackboard*. Through this, behaviour tree components can access the registered coordinates of the reference points.

The atomic *Manual Adjustment* element serves as a fallback to ensure that the arm can always be calibrated. The user must adjust the camera so that the reference points projected onto the camera image coincide with the real points. In other words, the user must find a predefined camera angle and distance to ensure valid coordinates.

After the *Camera* component completes successfully, the *Robotic Arm* component is executed, which is also sequential. Its first step is running the atomic *Connection Check* element, which is considered successful if the program communicates properly with the robotic arm. If the arm is not connected or is used by another program, an error is signalled.

Reference Point Registration is an atomic element requiring manual intervention, during which the real coordinates of the reference points are recorded with the help of the user. This is a simple process in which the end effector of the robotic arm must be placed on all six points from *A* to *F*. The height relative to the mat does not matter, as only the *X* and *Y* coordinates are relevant. Once the end effector is positioned on a reference point, the system queries and records the current coordinates of the end effector. At the end of the process, the scanned data are also stored in the shared *blackboard*.

The atomic *Height Setting* element is similar to the previous

one, except here only the height of the end effector matters. The user must set this value by adjusting the end effector, and the system records it. This is the default height level the arm will use during operations unless overridden. The set height value is communicated to the robotic arm. For both the *Reference Point Registration* and *Height Setting* components, error checking is performed to filter out values outside the workspace.

After successful execution of the *Camera* and *Robotic Arm* components, all conditions are met to run the *Projective Transformation* element. In this step, the mapping between the two planes is established using the method described above, and a transformation function is built that can convert coordinates between the two systems. Although at least four reference points are required for mapping, this does not guarantee that the transformation exists. The underlying system of equations may have no solution, for example, if the reference points are collinear [11]. In such cases, calibration must be repeated.

On the user interface, the calibration process includes an additional step for selecting the test device. However, this only sends feedback to the server and is not part of the behaviour tree, as it is always considered a successful operation.

IV. Tests

Software tests typically run in a virtual environment where all parameters can be precisely controlled. In contrast, hardware testing must account for physical factors such as mechanical elements of devices, which can present significant challenges. For all these tests, a structure had to be devised that can support testing for touchscreen or other hardware devices as well.

Some interaction elements (such as buttons, scrollers, touchscreen interfaces) provide the user with the opportunity to move from one state of the system to another. These states can be considered as different pages or user interfaces between which the user can move through a specific action. For example, by pressing the *Back* button, the user can return to a previous state from the current page, while a *Like* button does not change the navigation state, but performs a modification on the given page (for example, by updating the state of an element).

After studies, the idea emerged that the structure enabling navigation can be effectively represented in the form of a graph, regardless of whether a given operation actually results in a state change or merely performs a modification within the screen.

This approach enables structured mapping of the operation of applications and hardware devices, as well as helps define and execute automated tests. States (pages) can be interpreted as nodes of the graph, while interaction elements that perform various operations, such as navigation, button presses, or modifying settings, constitute the edges of the graph.

The graph-based representation not only enables structured mapping of the operation of applications and hardware devices, but also contributes to the efficient design and execution of automated tests. With this type of representation, testing paths can be precisely defined, potential errors can be more easily identified, and it can be ensured that every interaction is covered during the testing process.

TABLE II
Description of the edge object fields

Field	Description
`id`	The unique identifier of the edge, which allows for unambiguous identification.
`componentName`	The name of the interactive component found on the application interface.
`from`	The node (page) from which the navigation starts.
`to`	The node (page) to which the navigation occurs.
`static`	Boolean value: for static (true) edge there is no navigation, while for dynamic (false) there is.
`deltaHeight`	The parameter of the touch function that modifies the default execution height.
`push`	Boolean value: if true, pressing is required; if false, touching is enough.
`action`	The name of the interaction operation.
`direction`	The direction of the operation (e.g., `up`, `down` for scrolling).
`hold`	The duration of the touch in seconds.

To store the data in memory, structured JSON files are used. In this format, nodes (pages) and edges (interaction elements) can be easily defined, which help to accurately and comprehensibly represent the navigation tree of the application or system. This ensures that the system remains easily modifiable and expandable, while the data remains well-structured and simple to manage.

The tests that the user can create consist of steps. Each step represents edges in the graph, that is, operations that test the functionality of the given test device. For example, checking the correct operation of a button or examining the response of an interaction element may belong here. The sum of the individual steps constitutes the complete test. Since the steps of the tests are determined by the graph structure, they automatically fit into the navigation process of the system and provide an opportunity to test functionality.

The parameters stored in the JSON objects of the navigation graph contain not only the data that the user needs to know directly, but also other background information that is essential for the operation of the system but not necessary for the user.

There are two main object lists in the JSON file: *nodes* (nodes) and *edges* (edges). Within the *nodes* object list, other objects are stored that have an *id* and a *name* field, where the *id* represents a unique identifier, and the *name* is the name of the page within an application/interface. Within the *edges* object list, the edges are stored (see edge object fields in Table II).

The tests and their correctness conditions are received by the backend server in *JSON* format, which then checks and processes them.

Each test step has a behaviour tree assigned to it, which is

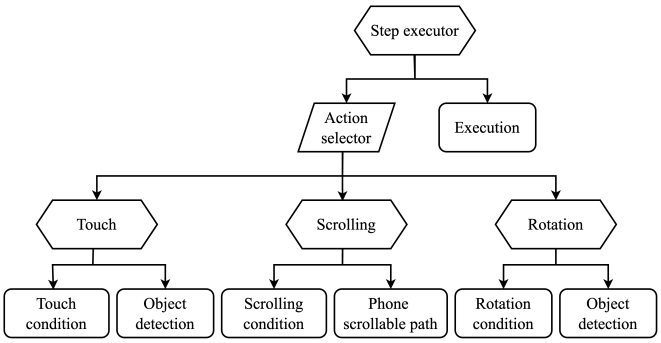

Fig. 4. Decision tree for the execution of steps

responsible for recognizing objects in the image provided by the camera and executing the desired operation (see Fig. 4).

The *Step executor* is a sequential component that runs all of its direct subordinate elements, the *Action selector* and the *Execution*. If any of these components do not execute correctly, the entire step is considered unsuccessful.

The *Action selector* is a selector component that stops after running the first successful direct subordinate element. This allows the system to decide which operation the given step belongs to.

The *Action selector* component has three subordinate elements: *Touch*, *Scrolling*, and *Rotation*. All three are sequential components, from which the system selects which operation to execute based on the supported operations, and also sets the necessary parameters.

During the execution of these operations, the *Touch*, *Scrolling*, and *Rotation condition* components can return success or error states, which decide which operation should continue.

For the *Touch* and *Rotation* operations, an interface component must be recognized on the test device. The *Object detection* node is responsible for this task, which sets the appropriate coordinates of the component using the so-called *blackboard* keys. These coordinates can be read by later nodes and used to execute the given operation.

In the case of *Scrolling*, it is not necessary to recognize a component, but rather to determine the position of the device, the so-called *Phone scrollable path*. This requires two coordinates that describe the path where the scrolling operation should be performed. Using computer vision, the frame of the phone screen can be outlined, and using distance analysis, the section where the user typically scrolls can be determined. To solve this task, the `fitLine` function of *OpenCV* is applied. In this case as well, the two coordinates needed to determine the section are stored using the *blackboard* keys.

After the *Action selector* has successfully completed the tasks necessary for the operation, the *Execution* node reads from the *blackboard* the scrolling coordinates (if they exist), and the component centre point coordinates, which were set by the *Object detection* nodes. For each operation, the appropriate functions belonging to the robot module are called.

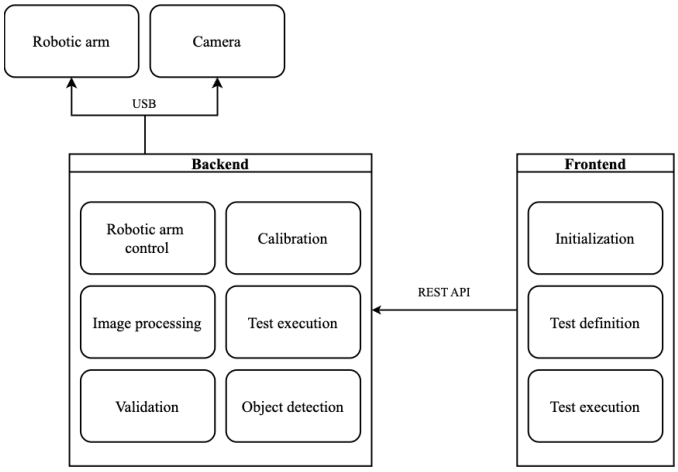

Fig. 5. System architecture

To check the correctness of tests, predefined conditions are needed that ensure the success of the test can be evaluated. These conditions allow the user to decide on a test result without reviewing the testing process on video or personally monitoring the execution.

The system provides the opportunity to create customized correctness conditions. The user can select those interface elements that must be present after the test execution, as well as those that should not appear. If the user does not define custom conditions, the default criterion for success is that at least 60% of the expected interaction elements should be recognizable after execution.

For other testing possibilities, *OCR* (optical character recognition) algorithms are applied, which are capable of extracting textual information based on images. The user can specify an expected text for each step that should appear on the screen of the device being tested after the step is executed. This provides an opportunity for the system to automatically check whether the device has actually entered the desired state. The best performing solution is *EasyOCR*, an open-source *Python* library that supports character recognition in multiple languages.

To increase the accuracy of text recognition, the images go through multi-step preprocessing before the *OCR* algorithm runs. The input image is first converted to greyscale, then the contrast is increased using the *CLAHE (Contrast Limited Adaptive Histogram Equalization)* method to better distinguish the text from the background. After that, a sharpening filter is applied, which further highlights the characters, and finally, the image is enlarged so that the *OCR* can work from a higher resolution.

V. Architecture

The application consists of two distinct components: a processing layer and a presentation layer. The processing layer contains all functionalities related to object detection and device control, while the presentation layer communicates via *HTTP* requests to display relevant features to the user (see

Fig. 5). During the execution of the processing layer, it is crucial to consider hardware resources, as this layer requires greater computational capacity due to model execution and multithreading. Thanks to the clear separation of layers, the presentation layer does not impose significant hardware requirements and can be easily deployed elsewhere.

Communication between the layers is handled via a *REST API*, with the exception of video streaming, which uses *sockets* for increased speed.

The processing layer (backend) is implemented in Python, as the *Dobot Magician Lite* must be programmed in this language, and state-of-the-art AI tools such as YOLOv8 are also available in Python. This layer also provides an *API* for the presentation layer, functioning as a backend server.

The presentation layer is a web interface served by a *Flask* server, because it offers all needed functionalities without unnecessarily burdening application performance. The robotic arm and camera are connected to the server exclusively via USB cable, which significantly complicates remote deployment.

The backend server provides numerous functionalities for the presentation layer, which are offered through an API following *REST* conventions, a logical and well-organized framework for endpoints. All endpoints use the convenient and natural JSON format for payload transmission.

Persistent data storage is handled by a *PostgreSQL* database, chosen for its efficient handling of JSON data and excellent compatibility with Python. The database includes tables such as *Tests*, *Configs* (loadable calibrations), and *Subjects* (test devices). The server can connect to a local or remote database, the location of which is configurable via environment variables.

The web application is written in *JavaScript* using the *React* library, resulting in an intuitive and clean interface. Since the application is not intended solely for experts, this is taken into account during interface design.

A designer is involved in the interface development, resulting in a design prototype created in the *Figma* design program, which serves as the basis for the website. The required *React* components are built by customizing elements provided by the *Bootstrap* library.

All functionalities of the web application are accessible exclusively to authenticated users. User management is provided by a private *Keycloak* server, made available for the project by *Codespring*.

VI. Conclusions and Development Opportunities

The system presented in this paper provides an opportunity for the user to test a device requiring interaction using a robotic arm, relying on object detection. The testing process can be accessed through a web application, which ensures access to the functions provided by the application. After calibrating the arm, the user is able to execute a multi-step test. The system offers individually specifiable verification conditions for each test and step, which automatically evaluate the success of the test.

The foundations of the system have proven to be well-usable; however, further developments are needed to increase reliability. In addition, several functionalities can be incorporated that enhance the user experience and bring it closer to a marketable state. The following development opportunities are noteworthy:

- Importing and exporting test configurations from a database, as well as creating a dashboard where the user can view previously run tests and their results.
- Creating statistics based on the executed tests and their results, such as success rate, common errors, average running times, etc.
- Providing the opportunity to run multiple tests consecutively and automatically.
- Developing a system that allows the user to introduce new test devices.
- Implementing the deployability of the backend server, as well as detaching the communication with the robotic arm from the wired connection.

References

[1] B. J. Dilworth, A. Karlicek, and L. Thibault, "An approach to component testing: An analytical study," *Sensors and Instrumentation, Aircraft/Aerospace, Energy Harvesting & Dynamic Environments Testing, Volume 7*, 2019.

[2] D. Piumatti, E. Sánchez, P. Bernardi, R. Martorana, and M. Pernice, "An efficient strategy for the development of software test libraries for an automotive microcontroller family," *Microelectronics Reliability*, vol. 115, p. 113962, 2020.

[3] J. Shi, W. Li, W. Wang, and L. Guan, "Facilitating non-intrusive in-vivo firmware testing with stateless instrumentation," *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[4] A. Ahmed, O. Mosbahi, M. Khalgui, and Z. Li, "Toward a new methodology for an efficient test of reconfigurable hardware systems," *IEEE Transactions on Automation Science and Engineering*, vol. 15, pp. 1864–1882, 2018.

[5] S. Takekoshi, T. Shinagawa, and K. Kato, "Testing device drivers against hardware failures in real environments," *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016.

[6] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, and R. Qu, "A survey of deep learning-based object detection," *IEEE Access*, vol. 7, pp. 128 837–128 868, 2019.

[7] R. Kaur and S. Singh, "A comprehensive review of object detection with deep learning," *Digital Signal Processing*, vol. 132, p. 103812, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1051200422004298

[8] M. Hussain, "YOLO-v1 to YOLO-v8, the Rise of YOLO and Its Complementary Nature toward Digital Manufacturing and Industrial Defect Detection," *Machines*, vol. 11, no. 7, 2023.

[9] T. Diwan, G. Anirudh, and J. Tembhurne, "Object detection using YOLO: challenges, architectural successors, datasets and applications," *Multimedia Tools and Applications*, vol. 82, pp. 9243–9275, 2023.

[10] Shenzhen Yuejiang Technology Co., Ltd, *Dobot Magician Lite User Guide (DobotLab-based)*, Shenzhen Yuejiang Technology Co., Ltd, Floor 9-10, Building 2, Chongwen Garden, Nanshan iPark, Liuxian Blvd, Nanshan District, Shenzhen, Guangdong Province, China, 2022. [Online]. Available: https://download.dobot.cc/product-manual/magician-lite/en/Dobot%20Magician%20Lite%20User%20Guide%20(DobotLab-based).pdf

[11] E. Dubrofsky, "Homography estimation," *Diplomová práce. Vancouver: Univerzita Britské Kolumbie*, vol. 5, 2009.