COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

MASTER'S DEGREE IN INDUSTRIAL ENGINEER AND SMART GRIDS

FINAL MASTER THESIS

DESIGN AND IMPLEMENTATION OF A TEST AUTOMATION STRATEGY FOR POWERON CONTROL SYSTEM

Autor: Alberto López-Rey Rojas
Director: Graeme Burt
Co-Director: Bruno Bicarregui Sánchez Sánchez

Madrid
Agosto de 2025

Design and Implementation of a Test Automation Strategy for PowerOn Control System

Alberto López-Rey Rojas, *Author*, Graeme Burt, *Director* and Bruno Bicarregui Sánchez Sánchez, *Director*

***Abstract*— This work presents the design and implementation of a visual test automation framework for PowerOn, the SCADA/ADMS platform used by Scottish Power. The challenge addressed is the automation of functional and regression testing in a closed, GUI-only environment without backend access or public APIs. After evaluating several tools, SikuliX was selected for its cost-effectiveness, open-source nature, and adaptability to pixel-based recognition. The framework interacts exclusively with the graphical interface, simulating operator actions through image matching, OCR, and keyboard/mouse inputs. It supports modular test design, adaptive similarity thresholds, and error-recovery mechanisms, with runtime configuration via Java Swing components. Batch execution enables scalability, running tests on large datasets while generating detailed CSV logs with outcomes, errors, and screenshots. A case study on the Network Diagram interface achieved a 94% success rate across 200 substations with a median runtime of 10.5s, demonstrating robustness, repeatability, and efficiency. Although commercial tools like Eggplant offer more advanced features, SikuliX proved sufficient for project goals under budget constraints. The framework provides a foundation for extending test automation to other PowerOn modules, integrating backend checks, and developing centralized reporting, contributing to more reliable and auditable testing in critical infrastructure systems.**

***Keywords*— PowerOn, GUI-based Automation, Test Automation Framework, SikuliX, Image Recognition, Optical Character Recognition (OCR), Critical Infrastructure Testing.**

I. INTRODUCTION

Modern power utilities increasingly rely on Supervisory Control and Data Acquisition (SCADA) and Advanced Distribution Management Systems (ADMS) to ensure network reliability, optimize daily operations, and comply with regulatory requirements [1], [2]. Among them, *PowerOn*, developed by GE Grid Solutions [3] and used by Scottish Power, is a mission-critical platform that integrates multiple modules such as Network Diagram, GeoView, and fault restoration systems. Operators depend on it for monitoring, control, and decision-making in real-time grid operations.

Testing in PowerOn has traditionally been conducted manually. Procedures involve expert operators executing long scripts, visually verifying results, and recording outcomes. This approach is slow, error-prone, and lacks scalability. The absence of backend access, public APIs, or inspectable object trees prevents the use of conventional automation frameworks (e.g., Selenium, Cypress), making automation in PowerOn a challenging task.

To overcome these limitations, alternative strategies based on graphical user interface (GUI) automation have been explored. Several tools exist in this domain, including Eggplant, Ranorex, TestComplete, Autolt [4], and SikuliX. Eggplant offers advanced model-based capabilities but requires costly licenses [5], while Ranorex and TestComplete depend on internal UI object access [6] [7]. SikuliX, on the other hand, is an open-source tool leveraging image recognition (OpenCV) and optical character recognition (Tesseract), making it particularly suited to closed, GUI-only environments such as PowerOn [8].

This work addresses these challenges by designing and implementing a modular, scalable, and low-cost visual test automation framework for PowerOn using SikuliX. The proposed framework simulates human operator actions through image pattern matching, OCR, and input simulation. A representative case study on the Network Diagram module validated its robustness and scalability, achieving a 94% success rate across 200 substations. The results demonstrate that GUI-based automation can be a reliable and efficient testing strategy in closed SCADA environments, providing a foundation for future extensions and integration into broader testing pipelines.

II. PROJECT SCOPE AND OBJECTIVES

This project aims to design and implement a visual test automation framework for PowerOn, a critical SCADA/ADMS system used by Scottish Power for network control and management. Due to the platform's lack of backend access, absence of public APIs, and restricted internal architecture, testing must be conducted entirely through its graphical user interface. This presents a unique challenge for test automation, requiring a solution that can simulate human interaction with the GUI in a reliable, repeatable, and scalable manner.

The scope of the work includes both technical development and methodological research. From a technical perspective, the project involves creating modular automation scripts using SikuliX to replicate essential test procedures already performed manually within the RTS (Real-Time Systems) department. From a research perspective, the work involves evaluating various tools, benchmarking their capabilities, and critically assessing their applicability to closed systems like PowerOn.

The automation framework developed as part of this project focuses on a selection of key user workflows that are representative of the broader system. While the Network Diagram and GeoView interfaces were initial targets due to

their visual complexity and frequency of use, the framework was designed to support multiple PowerOn interfaces, including Work Package Manager, Safety Documents, Incident Management… This diversity introduces varying visual layouts and interaction behaviours, which the automation logic had to accommodate.

The framework includes functionalities:

- Search for and open location-based records (such as substations, switching points, fault locations) across different PowerOn modules, like Network Diagram or GeoView.
- Apply and validate multiple categories of filters depending on the requirements of each test, including sensitive case, visible locations, low voltage locations, location type (transmission, sub transmission, grid, primary, secondary) or location sub type (substation, switchgear, minor). Depending on the interface context each filter will be applied or not.
- Interact with dynamic toolbars, navigation panels, popup dialogs, and search fields, while adapting to different screen states and UI configuration.
- Visually validate system state changes using image comparison and OCR. For example, confirming a selected element is highlighted, a scale has been applied, or a message window has been dismissed.
- Automatically log test results in a structured and repeatable way, capturing pass/fail outcomes, timing metrics, and specific visual errors.

To support both scalability and data collection, the project also includes functionality to run the same test flow across a batch of substations (potentially hundreds) and record all results in a structured CSV file. This feature not only enables efficiency and repeatability but also allows the company to build test evidence and measure reliability over time.

Importantly, the project is not intended to replace the full testing strategy of the RTS department, but to provide a proof-of-concept and a scalable foundation for future automation. The focus has been placed on building a robust, maintainable system that could be expanded with minimal effort in future phases, either by integrating with test reporting systems or by extending the test library to cover more scenarios.

The objectives were defined to ensure meaningful progress while acknowledging realistic constraints. As such, only selected test scenarios were developed and validated, and testing was limited to environments provided by Scottish Power without full production access.

In summary, the key project objectives can be stated as follows:

1. To investigate and compare available tools for automating GUI-only systems like PowerOn, and to select the most suitable one given technical and budgetary constraints.
2. To design a modular and extensible visual automation framework using the selected tool (SikuliX).
3. To implement and validate test procedures based on existing manual testing workflows in the RTS department.
4. To enable batch testing and automatic result logging for a wide range of substations.
5. To provide a structured and well-documented foundation for future expansion and potential integration into broader testing infrastructure.

These objectives were used to guide the methodology, implementation, and evaluation of the project, ensuring alignment with both academic requirements and business needs.

The following chapter reviews the state of the art in test automation for closed systems, providing the technical and industrial context that informed these objectives.

III. Methodology

The automation framework was developed within the PowerOn test environment provided by Scottish Power, which replicates operational conditions while restricting backend access and administrative privileges. This constraint required all interactions to occur through the graphical user interface, making visual automation the only viable approach.

A. Tools and Environment Setup

All experiments were carried out in the PowerOn test environment (v6.9.3) provided by Scottish Power. This environment replicates the operational platform used in real distribution networks while limiting administrative access and backend connectivity. Consequently, all automation had to be performed exclusively through the graphical user interface.

The main tool selected was SikuliX 2.0.5, an open-source automation framework that leverages OpenCV for image pattern recognition and **Tesseract OCR** for optical character recognition. SikuliX enables scripts to detect on-screen elements, simulate mouse and keyboard interactions, and validate textual information displayed on the interface. Scripts were implemented in Jython, which provides a Python-like syntax on the Java Virtual Machine, facilitating modular design and integration with other components [8]. Additional utilities included:

- **Java Swing:** To improve usability and reduce user error during test execution, several interface components were implemented using Java Swing, which is supported natively within the SikuliX

environment. These elements provided a basic user interface layer on top of the script logic, allowing operators and testers to interact with the automation in a structured and guided way, without modifying the code or using external tools.

- **CSV-based datasets:** Structured CSV files were used to manage test inputs (such as location names, interface elements, or operational parameters provided by Scottish Power) and to store output logs. This approach allowed the framework to dynamically load real-world test data while keeping the execution process consistent and automated. It also simplified the user's role, making it possible to prepare large test batches without altering the source code. Crucially, this structure enabled the same automation logic to be executed repeatedly across a wide range of varied input cases, with outputs logged for each run. This approach supports efficient large-scale validation and allows results to be compared, analysed, and audited over time.
- **Screenshot logging:** to capture visual evidence of error or inconsistencies during execution.

This toolchain was selected after evaluating alternatives such as Eggplant, Ranorex, TestComplete, and Autolt. While Eggplant offers advanced model-based testing, its high licensing cost (€10,000 approx.) and proprietary language (SenseTalk) made it unfeasible within the project scope. SikuliX, although more sensitive to UI variability, was sufficient to demonstrate the viability of GUI-based automation in PowerOn.

In Table 1 a consolidated overview of the key strengths and weaknesses or each automation tool is shown. The scoring is based on both technical documentatiton and hands-on experimentation within the constraints of the PowerOn environment.

From Table 1, it becomes clear that Eggplant scores highest in advanced capabilities such as AI-driven testing, OCR accuracy, integration, support, and compliance, making it the most robust solution overall. However, its low score in cost (due to its expensive commercial license) and relatively steeper learning curve in custom scripting make it a challenging choice for lightweight or budget-constrained projects.

SikuliX, while not leading in raw power or AI features, achieves the best balance of visual automation, scripting flexibility, cost-efficiency, and suitability for PowerOn. It scored particularly high in categories like cost (5/5), custom scripting (5/5), and PowerOn compatibility (5/5), which reflects its adaptability to GUI-only environments where no internal access or API integration is possible. Its open-source nature, lightweight installation, and compatibility with CSV or excel inputs and Java GUI elements make it ideal for academic research and proof-of-concept development.

Table 1. Tool Criteria Scores

Tool	Cost	Installation	Visual Automation	AI Capabilities	Integration	OCR Accuracy	Documentation	Custom Scripting	Testing Flexibility	Scurity/ Compliance	Support	Suitable for PowerOn	AVERAGE
Sikulix	5	4	4	1	3	3	4	5	4	3	3	5	3,7
Eggplant	1	3	5	5	5	5	4	3	5	5	5	4	4,2
Autolt	5	5	2	1	2	1	3	3	2	2	2	2	2,5
Ranorex	2	3	3	2	4	3	4	4	4	4	4	1	3,2
TestComplete	2	3	3	3	4	4	4	4	4	4	4	1	3,3

Overall, the comparative evaluation underscores a key trade-off: Eggplant offers superior capabilities, but SikuliX is the most viable under real-world constraints.

B. Framework Architecture

The automation framework was designed with a **four-layer modular architecture** (**Figure 1**), each responsible for a specific set of tasks:

1. **Input Layer:** Parameters can be provided interactively via Java Swing components (manual mode) or imported from CSV files (batch mode). This allows the same workflow to be executed across hundreds of substations with no code modifications.
2. **Control Layer:** Orchestrates the flow of each test, including conditional branching, retries, and fallback logic. Watchdog mechanisms are embedded to automatically dismiss unexpected pop-ups and maintain continuous execution. [9] [10]
3. **Action Layer:** Executes GUI interactions such as clicks, typing, scrolling, and drag-and-drop. It also performs OCR-based checks to validate textual outputs (e.g., confirming whether a substation name or state change is correctly displayed).
4. **Output Layer:** Generates structured logs, storing execution results in CSV format along with timestamps, error descriptions, and screenshots. This evidence provides traceability and facilitates post-execution audits.

The modular design simplifies test maintenance and enables the framework to adapt to changes in the PowerOn interface. Image libraries were organized by module and function, with multiple fallback variants for each element to account for resolution differences or UI changes.

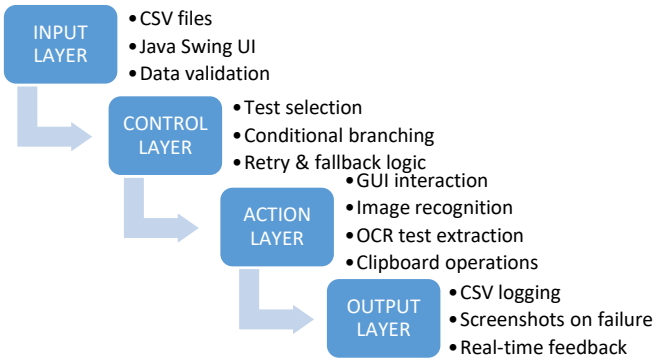

***Figure 1**. High-level architecture of visual automation framework*

C. Image Recognition and Robustness Mechanisms

Given the nature of PowerOn, a closed SCADA/ADMS platform without programmatic access, APIs, or an inspectable object tree, all automation tasks in this project were executed using image-based logic. This approach, while fundamentally different from DOM (Document Object Model) or code-based automation, is particularly well suited to applications where only the visual output is available to the tester. In such cases,

the screen itself becomes the only "interface" the automation can observe and act upon.

Every interaction in the framework is based on the principle of pattern matching, where the system searches the screen for a visual element (such as a button, label, icon, or panel) that matches a previously captured reference image. Once identified, actions can be executed (such as click the region, reading test from it, or waiting for its appearance or disappearance).

To ensure consistent and reliable detection, several best practices were implemented:

- **High-quality image capture:** the entire automation strategy is based on visual pattern matching, the quality, consistency, and reproducibility of reference images play a crucial role in system reliability. Every image used for matching was captured manually from the PowerOn test environment using the exact same conditions as those under which the automation would later run. Capturing high-quality and consistent images significantly reduces maintenance, lowers test flakiness, and made the test scripts more portable across machines with similar setups. If tests are to be shared across different environments in the future, a dedicated image calibration step or image regeneration tool could be developed to ensure alignment across systems.
- **Structured image storage:** To manage a growing library of visual elements used in different automation scenarios, a clear and scalable storage structure was designed. Instead of placing all reference images in a single directory, the image assets were categorised into a hierarchical folder structure, grouped by both interface context and functional purpose. This modular approach offer benefits as maintainability (developers can easily identify which tests use which images and update them with PowerOn evolving), scalability (structured storage prevents disorganisation and duplication, making it easier to navigate the library one the number of automated test cases grows) [11] [12], reusability (shared assets can be reused across different test scripts) and debugging and refactorings (when a tests fails due to an image mismatch, its location in the directory gives immediate context).
- **Fallback logic:** Given that PowerOn may display slightly different versions of the same icon or interface element depending on factors such as user settings, resolution, screen scaling, or even system theme, the automation framework had to account for visual variability. To address this, each critical UI element was assigned not one, but multiple reference images, which represent known visual variants of the same function. The automation code uses fallback logic, checking for all possible versions of a button or field before deciding that the element is not available. This was implemented using commands like *exists()* or *wait()* in chained conditions, trying several image matches sequentially with a fallback priority.
- **Similarity tuning:** The similarity parameter in SikuliX controls the tolerance with which the system matches a reference image to a region on the screen. Contrary to a uniform threshold, this project employed adaptive similarity tuning, where the required similarity level depends on the type of visual element and its functional context. Adaptative similarity thresholds were applied (0.3-0.99), allowing flexible matching depending on visual complexity. This adaptive approach reduced false negatives in flexible interfaces while still maintaining accuracy where needed. However, it also required extensive experimentation and manual tuning. Some UI components had to be tested with multiple similarity levels to find the optimal balance between robustness (avoiding test failures due to pixel-level differences) and precision (avoiding misidentification of unrelated screen content). This is shown in **Figure 2**, where matching setting is selected to be 0.35 and it recognise the elements, being a perfect example of how low similarity thresholds are some times more appropriate than higher ones.

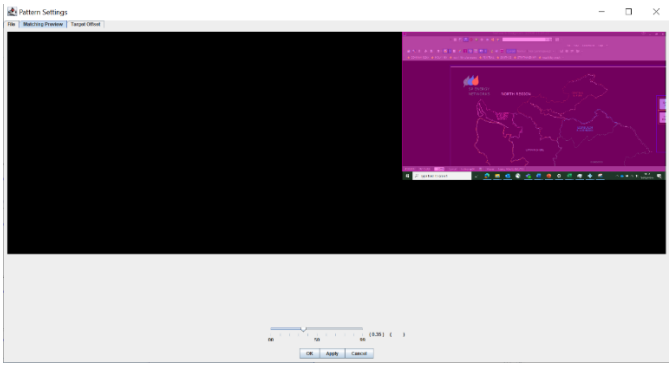

***Figure 2**. Matching settings example*

- **Sequential state validation:** the automation framework uses sequential state validation, combining multiple visual checks to confirm that the system is truly ready. This typically involves verifying that one expected image exists (e.g., a menu icon appears), while simultaneously ensuring that another known transitional image vanishes (e.g., a loading symbol or transition window disappears). These checks are implemented with *wait()* and *waitVanish()* calls in tandem, often within controlled timeouts. This mechanism proved essential for avoiding premature clicks or faulty reads, particularly views where the screen contents update dynamically. It also adds resilience in less predictable environments, ensuring the automation reacts to real screen conditions instead of relying on fixed delays.
- **Timeout strategies:** SikuliX scripts interact with the live screen and depend on the rendering speed of the system, each image detection action is paired with a defined timeout window. This prevents the automation from hanging indefinitely in cases where an expected element fails to appear due to error, delay, or user interference. Timeout durations are chosen based on the expected load time of each interface component, going from 0.5 seconds to 2 minutes. If the timeout

expires and the image is still not found, the frameworks would continue to the next item if the step is optional or it is batch mode, or halts the execution and displays a popup message, like in **Figure 3**.

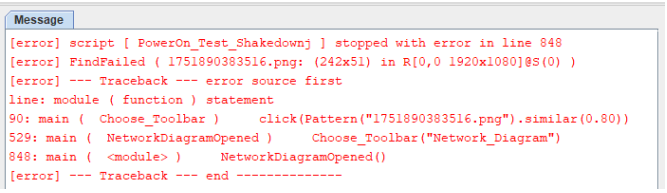

***Figure 3**. Image not found error message*

D. Batch Execution and Scalability

One of key strengths of the developed framework is its ability to execute test procedures across a large set of input values, in a consistent and fully automated manner. This is achieved through the use of CSV-driven batch testing, a mechanism that allows the same test logic to be dynamically applied to dozens or even hundreds of individual elements, such as locations, assets, configurations, or interface cases, without requiring manual intervention between test cycles.

The approach is based on a simple but powerful idea: separating the test logic (what actions to perform) from the data (on which elements to perform them). Input data is stored in structured CSV files, typically containing a list of items (e.g., names of substations, diagram references, configuration identifiers). These files are loaded at runtime, and the framework automatically iterates through each row, executing the selected test sequence and recording the outcome of each run.

The architecture provides several advantages:

- **Scalability:** the CSV-driven input model offers inherent scalability by decoupling test logic from the data it operates on. Testers can easily expand the coverage of a test suite simply by adding new entries to a CSV file, without altering the code or requiring programming knowledge. For example, if a new set of locations or interface components need to be validated, they can be appended to the input list and automatically included in the next test run [11] [12]. This means the system is capable of growing organically with minimal technical overhead, adapting to the evolving needs of business.
- **Reusability:** the automation framework separates "what to do" from "what to test it on", the same functions can be applied to different datasets, test modes, or workflows without duplication. This dramatically reduces development time and supports the principle of test case reusability.
- **Parallelisation potential:** this version of the framework executes test cases sequentially; the architecture is naturally suited to parallel execution in future iterations. Since each test cycle is independent (reading one row from the input CSV, executing a standard procedure, and writing the result) it can be parallelised across multiple processing threads, test agents, or even virtual machines. This opens the door to significant performance gains in large-scale deployments. For instance, testing 500 inputs that currently take several hours in sequence could be distributed across five agents and completed in a fraction of time. Additionally, parallelisation could support continuous testing in CI/CD pipelines or enable regression testing on large datasets in an overnight cycle. Preparing the system for this future capability reflects a forward-looking design focused on industrial scalability.
- **Result granularity:** The logging system was designed to offer fine-grained visibility into the performance and stability of each individual test case. For every data entry processed, the framework writes a row to a structured output file, detailing not only whether the test passed or failed, but also why, when and how it was processed. This includes fields such as: input item name, test category, result statues, error type, execution timestamp, screenshot or reference for debugging. Such granular output enables a range of post-test analysis activities: error trend detection, coverage analysis, historical comparisons, and auditing. It also facilitates transparency and accountability, allowing to understand not just what failed, but where and why, paving the way for rapid debugging and system improvement. The Figure 4 shows an example of the results for Network Diagram substation test, where the fields commented are included and there are some examples of error messages.

Substation	Result	Error	Failed Step	Screenshot	Durati	Timestamp
AYR GSP	Pass				15.432	10/08/2025 11:18
BONNYBRIDGE GSP	Pass				9.043	10/08/2025 11:18
CARNTYNE GSP	Pass				9.141	10/08/2025 11:18
CHAPELCROSS GSP	Pass				9.085	10/08/2025 11:18
CHATLOTTE ST G.S.P.	Pass				9.415	10/08/2025 11:19
COATBRIDGE A GSP	Pass				9.833	10/08/2025 11:19
COYLTON GSP NEV	Pass				9.915	10/08/2025 11:19
CROCKSTON GSP	Pass				10.407	10/08/2025 11:19
CUMBERNAULD GSP	Pass				9.795	10/08/2025 11:19
DALMARNOCK GRID	Pass				9.945	10/08/2025 11:19
DEVOL MOOR GSP	Pass				9.885	10/08/2025 11:20
DRUMCHAPEL GSP	Pass				10.322	10/08/2025 11:20
DUMFRIES GSP	Pass				10.181	10/08/2025 11:20
DUNFERMLINE GSP	Pass				10.531	10/08/2025 11:20
EASTERHOUSE GSP	Pass				9.243	10/08/2025 11:20
ELDERSLIE GSP	Pass				9.899	10/08/2025 11:20
ERSKINE GSP	Pass				9.956	10/08/2025 11:21
FINNIESTON	Fail	[Open scale] Cannot click; not found. 1751892054482 Open scale		C:\Users\B620717\Desktop\PowerOnResults\snap_auto_fail_20250810_112116.png	13.1	10/08/2025 11:21
GLENCLUCE GSP	Fail	[Open filters] Cannot click; not found. 1752064695710 Open filters		C:\Users\B620717\Desktop\PowerOnResults\snap_auto_fail_20250810_112123.png	6.585	10/08/2025 11:21
GLENLUCE GSP	Pass				16.15	10/08/2025 11:21
GRANGEMOUTH A GSP	Pass				9.099	10/08/2025 11:21
GRANGEMOUTH C GSP	Pass				9.824	10/08/2025 11:22
GREENOCK GSP	Pass				9.63	10/08/2025 11:22
HAGGS RD GSP	Pass				9.503	10/08/2025 11:22
HELENSBURGH GSP	Pass				9.335	10/08/2025 11:22
HUNTERSTON FARM GS	Pass				9.176	10/08/2025 11:22
KILBOWIE GSP	Pass				9.667	10/08/2025 11:22
KILLERMONT GSP NEV	Pass				9.25	10/08/2025 11:22
KILMARNOCK SOUTH GS	Pass				9.355	10/08/2025 11:23
KILWINNING GSP	Pass				9.259	10/08/2025 11:23
LINNMILL GSP	Pass				9.212	10/08/2025 11:23
LIVINGSTON EAST GSP	Pass				9.287	10/08/2025 11:23
MAYBOLE GSP	Pass				9.395	10/08/2025 11:23
NEVARTHILL GSP (NEV	Pass				9.24	10/08/2025 11:23
NEWTON STEVART GSP	Pass				9.084	10/08/2025 11:24
PARTICK GSP	Pass				9.55	10/08/2025 11:24
RAVENSCRAIG GSP	Pass				9.097	10/08/2025 11:24
SALTCOATS B	Pass				9.21	10/08/2025 11:24
ST ANDREWS CROSS G	Pass				9.464	10/08/2025 11:24
STIRLING GSP	Pass				8.981	10/08/2025 11:24
STRATHAYEN GSP1	Pass				9.162	10/08/2025 11:24
STRATHLEVEN GSP	Pass				9.175	10/08/2025 11:25
TONGLAND GSP (T)	Pass				9.197	10/08/2025 11:25
WEST GEORGE ST GSP	Pass				9.962	10/08/2025 11:25
WESTBURN ROAD GSP I	Pass				9.639	10/08/2025 11:25
WESTBURN ROAD GSP	Pass				9.594	10/08/2025 11:25
WHISTLEFIELD GSP	Pass				9.661	10/08/2025 11:25
VISHAV GSP	Pass				9.43	10/08/2025 11:26
VISHAV GSP A	Pass				8.579	10/08/2025 11:26
VISHAV GSP B	Pass				9.458	10/08/2025 11:26

Figure 4. Results CSV example

IV. Case Study: PowerOn Network Diagram Module

To illustrate the application of the architecture and design principles described in the previous sections, this chapter presents a complete end-to-end test case developed as part of the automation framework. This practical example demonstrates how the system executes a structured series of actions to validate interface behaviour, interact with graphical elements, handle system variability, and log the result of each test cycle.

The scenario chosen for this case involves the visual verification of an electrical location within PowerOn's Network

Diagram and GeoView interfaces. This test mirrors a real-world validation procedure frequently performed manually by RTS operators, and was selected because it includes all major aspects of the automation pipeline; input collection, interaction with dynamic UI components, error handling, and result logging.

A. Automated Workflow

The automated workflow followed ten steps:

1. **Launch and Log in:** Before the automation can begin, the user must manually open PowerOn and log in using their own credentials. This manual step is required due to system security policies and access controls: login screens are outside the scope of GUI automation in this environment, and credentials cannot be passed or injected via scripts. The framework assumes that PowerOn is running, authenticated, and stable, with all critical UI components fully loaded. Failing to meet this precondition may result in early test failure or missed elements.
2. **Collect test input:** Once PowerOn is ready, the framework begins by collecting the target location or test item. This is done in two ways, depending on the operating mode. In manual mode, a popup input dialog (created using Java Swing) prompts the user to enter the name of the asset or location to test, typically a substation, node, or operational point, such as shown in Figure 5. In batch mode, the system loads a list of entries from an external CSV file, iterating through each one in turn. This method is used for large-scale testing or regression validation.

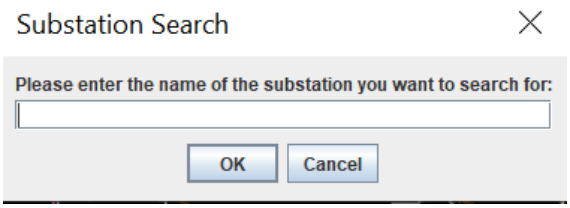

Figure 5. Popup dialog for manual testing

3. **Verifying toolbar access:** PowerOn toolbar is essential for accessing different modules (Figure 6**Error! Reference source not found.**). The automation begins by checking whether it is currently visible on the screen. If not, the script looks for the toolbar symbol by using a library of pre-captured images. If found, it clicks the symbol to open the toolbar. This step guarantees a predictable UI layout and prevents accidental interference with navigation or pattern matching later in the workflow.

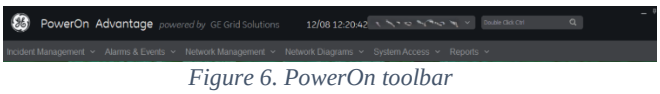

Figure 6. PowerOn toolbar

4. **Open Network Diagram interface:** once the toolbar is accessible, the script opens the Network Diagram module by clicking its corresponding button. After the initial action, the system waits for several visual conditions to confirm that interface has fully loaded (as in Figure 7), no overlay windows remain visible, and the view is maximised.

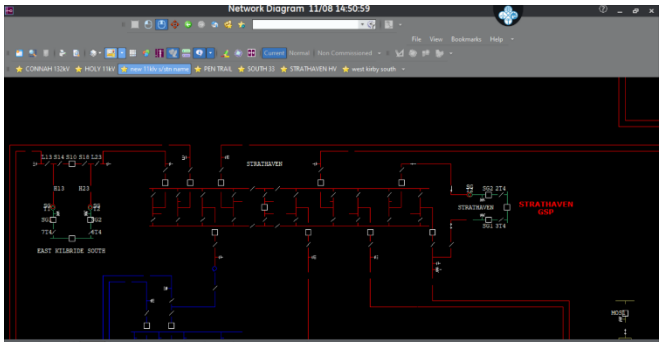

Figure 7. Network Diagram

5. **Prepare diagram environment:** before performing the search, the system applies a predefined set of diagram filters, with the settings shown in . To ensure only relevant assets are shown the following options are enable or not: voltage levels (low or high voltage), location types (transmission, sub transmission, grid, primary or secondary) and visibility filters and search modes.

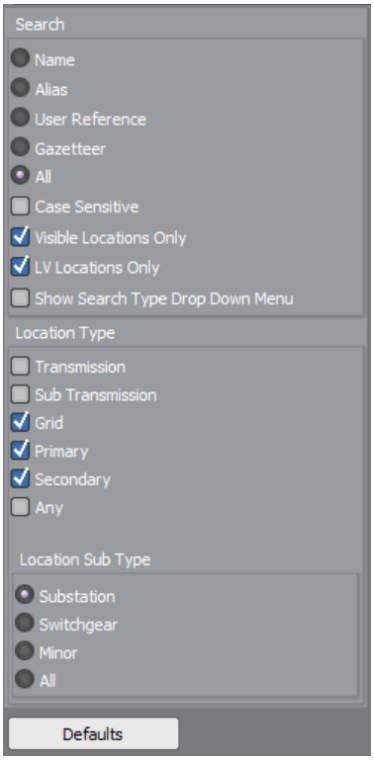

Figure 8. Filter settings

6. **Search for the location:** the test item (e.g., substation name) is entered into the search field of the interface. To improve consistency and execution speed, the framework uses a function called *copy_to_clipboard(Text)* that takes the location name, copies it to the clipboard, and pastes the value; avoiding the latency and inconsistency of simulated keystrokes. After this, the system clicks to initiate the search and waits for the result panel to appear. If

multiple results are shown (it should not happen if everything went well during the script), it selects the first candidate in the list. This behaviour ensures the test always proceeds with a matching, recognisable entry, and avoids ambiguity when search results are partially matched.

7. **Verify display and scale:** Once the location is selected, the automation adjusts the zoom level or visual scale of the diagram to a standard value (a scale of 1.6 was selected in the script), ensuring uniform rendering across test runs and simplifying pattern recognition.
8. **Switch to GeoView and Repeat validation:** once the Network Diagram test is completed, the system closes the module (to ensure the next test works well), then the toolbar is opened (if closed) and switches to the GeoView module. All actions described previously (filtering, searching, visual validation) are repeated within this different interface context. Although the overall workflow is similar, the visuals and rendering logic in GeoView differ (see in Figure 9), requiring separate reference images and detection logic. The framework handles this by using mode-specific image sets and conditionals to execute the correct steps.

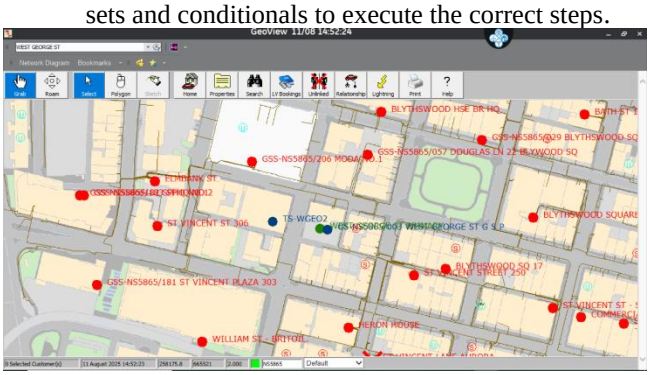

Figure 9. GeoView Diagram

9. **Log the result:** after completing the full test cycle, the system evaluates the success or failure of each major step. A new output CSV file is created with test inputs (e.g., location name), test mode (e.g., Network Diagram, GeoView Diagram), result (pass or fail), optional error message, timestamp of execution and screenshot of failure state. This logging mechanism provides traceability, debugging support, and enables data analysis for performance monitoring or future refinement.
10. **Batch mode execution:** if test runs in batch mode, the automation restarts the loop, pulls the next input from the list, and repeats the process. This continues until all entries in the CSV have been processed. Batch mode is typically used when validating many elements across the network or during regression testing phases to ensure no changes have broken expected functionality.

B. Results Log and Report

A critical component of any test automation framework is the ability to record, store, and interpret results in a structured and accessible way. In this project, results logging was designed with a strong emphasis on traceability, reusability, and future integration, providing value to the operational testing needs of Scottish Power.

The system implements a custom CSV-based reporting mechanism that captures the outcome of each test iteration, whether triggered manually or executed in batch mode. This output serves as both a technical artefact for developers and an operational report for testing teams, enabling the evaluation of test coverage, identification of system inconsistencies, and long-term reliability tracking.

Some important aspects from these results are:

- **Structure of the result log:** Each execution of a test, whether successful or failed, triggers the creation of a detailed log entry in a structured CSV file. This approach was chosen for its wide compatibility, human readability, and seamless integration with data analysis tools such as Microsoft Excel, Power BI, Python, and SQL-based reporting systems. The structured of the result log was carefully designed to strike a balance between simplicity and expressiveness, ensuring it could be used both for immediate inspection by testers and for programmatic parsing in future analysis pipelines. Each row in the CSV corresponds to a single test iteration, including "Tested item", "Test category", "Test subcategory", "Result status", "Timestamp", "Error description", and "Screenshot path".
- **Execution feedback during runtime:** While structured result logging ensures long-term traceability and post-test analysis, it is equally important for the automation framework to provide real-time execution feedback to the user. This immediate layer of communication allows the tester or operator to monitor progress, detect anomalies early, and feel confident that the system is performing as expected, especially during long or complex test sessions. The feedback mechanism implemented in this project include several modalities, each serving a different purpose within the user interaction spectrum:
  - **On-Screen popup message:** Throughout execution, the system generates interactive popup messages using Java Swing components.
  - **Printed console logs:** In addition to graphical notifications, the system outputs real-time execution traces to the console or terminal window.
  - **Execution counters and batch progress indicators.**
  - **Error triggers and immediate alerts:** If a critical failure or unhandled exception occurs, the framework is configured to raise an immediate alert via a modal popup window.
- **Support for batch aggregation:** One of the major benefits of implementing a structured and row-wise result logging system is the ability to perform data aggregation and statistical analysis across large sets of test executions. Since each row in the output CSV

represents a unique test case (including metadata such as input value, test category, outcome, and timestamp), this structure naturally enables both quantitative performance evaluation and qualitative insight generation.

- **Design for auditability and expansion:** To ensure that the framework remains reliable and maintainable over time, it was designed with auditability, traceability, and extensibility as foundational principles. All test logs are generated in a standardised format and stored in clearly defined directories, with consistent naming conventions and timestamp-based organisation, making easy to locate and retrieve past test results, compare outputs across test cycles, identify historical trends in failure or instability, submit results as part of regulatory compliance or internal quality audits. Each log file represents a verifiable record of system behaviour, including both technical execution data and human-readable summaries. This traceability is especially important in critical infrastructure environments like SCADA/ADMS systems, where automated testing must meet high standards of transparency and documentation.

C. Limitations and Considerations

While the automation framework developed in this project successfully demonstrates the feasibility and scalability of GUI-based testing for closed SCADA systems like PowerOn, it is important to recognise its technical, operational, and strategic limitations. Understanding these boundaries is essential for setting realistic expectations, identifying areas for improvement, and guiding future development efforts.

The most important limitations observed are:

- **Dependence on visual environment:** The core limitation of the system lies in its dependency on visual pattern recognition. Since PowerOn does not provide APIs or direct access to its underlaying data structures, all interactions are performed through screen-based logic using SikuliX. While the system accounts for this using fallback images and similarity tuning, this reliance on pixel-level accuracy makes it more fragile than API-based test automation approaches.
- **Image maintenance overhead:** Because the framework operates entirely via screenshot-based logic, any updates to PowerOn's graphical interface may require updating the associated image library. This creates a maintenance cost proportional to the number of test cases and images used. Over time, unless managed properly, this can affect scalability.
- **Limited to observable behaviour:** One of the inherent constraints of visual automation frameworks is that they can only operate on observable screen states. This means that all test validations are restricted to what can be seen, captured, and interpreted through the graphical interface, excluding any form of internal or hidden system logic. This limitation does not make the tool less useful; it simply defines its scope. The framework is designed to test functional, UI-level behaviour, ensuring that what the user sees and interacts with behaves correctly. However, it is not a substitute for internal QA processes, such as unit testing, integration testing, or database validation.
- **Performance variability and system load:** The performance of PowerOn is affected by a range of runtimes variables: the size of the dataset being handled, the number of concurrent users, server response times, and the graphical load of rendering complex diagrams. As the automation relies entirely on-screen response and image matching, these factors can cause the execution to behave inconsistently or fail if not handled carefully. Large diagrams may load slowly than anticipated, causing timeouts during detection. System lag or dropped frames can cause certain elements to appear partially or with visual artefacts, leading to recognition failures. Popups or windows may not close cleanly, leaving overlays that interfere with image matching. This variability introduces some uncertainty in result interpretation. A test marked as "Fail" may not reflect a functional problem in PowerOn, but rather a temporary delay or detection miss. For this reason, failed test should be reviewed alongside logs and screenshots before drawing conclusions.
- **Manual preconditions and operator involvement:** Although much of the system has been automated, the current version still requires a limited degree of manual intervention, particularly at the start of a test session. This is due to a combination of security policies, PowerOn's access architecture, and practical constraints of the testing environment. Specifically:
    - User must launch PowerOn manually and enter their credentials, as the login interface is protected against automation and includes unpredictable graphical security elements.
    - Some UI elements require manual dismissal if they are not recognised by the watchdog logic.
- **Lack of advanced AI integration:** Although some commercial tools like Eggplant incorporate machine learning to improve test robustness, the current implementation using SikuliX is rule-based and deterministic. While this offers clarity and control, it limits adaptability in complex or ambiguous scenarios.

V. Result Analysis

The batch execution campaign for *Test 1 (Network Diagram)* was performed using Scottish Power's dataset which contains the information about the substations. The primary objective was to evaluate the operational stability, accuracy, and efficiency of the developed automation script when run repeatedly and under varying data inputs. By automating the same functional scenario across all dataset entries, the test aimed to replicate a realistic operational workload, identifying potential weak points in performance or recognition reliability.

Due to time constraints and the scope of this stage of the project, the batch execution was carried out exclusively for this

single test case. This decision was made because this test serves as a representative example of how the automation framework operates: it includes multiple user interface interactions, search operations, filtering steps, and verification stages. Successfully validating this test in batch mode provides sufficient insight into the framework's expected behaviour for other tests, while keeping the workload manageable within the available timeframe.

The execution process was designed to log the outcome of each iteration, capturing result, error message, failed step, screenshot, execution duration and timestamp.

This information was stored in a CSV results file generated dynamically for each execution session, ensuring that repeated runs did not overwrite previous data. The output file naming convention included a time stamp, allowing easy traceability of historical runs.

A. Execution Summary

The dataset consisted of 200 independent runs of the Network Diagram test on distinct substations, each representing a unique input scenario for the automation sequence. In total 188 out of 200 runs succeeded (94%). Median execution time per run was 10.15 seconds. The principal failure modes were "Click scale – not found" (6/12) and "Open filters – not foud" (6/12). Every iteration involved:

1. Launching and configuring the Network Diagram interface.
2. Applying the appropriate filters.
3. Search for the specific substation.
4. Confirming its presence and accessibility in the interface.
5. Recording results into the CSV file.

The result obtained were:

Table 2. Resuls summary

Total tests	Successful executions	Failed executions	Average execution time	Average successful time
200	188 (94%)	12 (6%)	9.79 s	10.15 s

The performance metrics suggest that the automation process consistent across different inputs, with no significant increase in execution time caused by data variability.

B. Success Rate Interpretation

The observed success rate of 94% across 200 executions provides statistically grounded evidence that the framework can reliably interact with PowerOn's GUI for Network Diagram searches under varied operational conditions.

The 94% pass rate is an encouraging indicator of the robustness and stability of the developed script. It confirms that the implemented logic, image recognition thresholds, and interaction timing are well-adjusted for the majority of operational cases. The low failure rate also suggests that the framework is sufficiently resilient to minor interface response delays or changes in display conditions.

It is also relevant to highlight that during execution, the automation successfully navigated through multiple interface states and visual variations without requiring any manual intervention. This includes handling differences in the location of UI elements, window sizes, and background interface data, factors that are often a challenge in SCADA and other closed-system environments.

C. Failure Analysis

All 12 failures fell into two image-recognition categories, as show in Table 3**Error! Reference source not found.**. These issues are actionable via: regenerating higher-quality image templates at 100% DPI, raising similarity thresholds for icons while using sequential state validation and introducing short fallback searches (alternative icon variants) before failing.

Table 3. Error messages from batch test

Step Affected	Error Message	Likely Cause
Open scale	Cannot click; not found: 1751992054822.png	Temporary delay in rendering or partial obstruction of the target UI element.
Open filters	Cannot click; not found: 1752064695766.png	Minor interface variation or incomplete load of the filter menu before interaction.

In both instances the script's built-in error logging correctly identified the step name, exception message, and captured a screenshot of the screen at the moment of failure. This capability is critical for post-mortem debugging and prevents silent or undetected errors.

Notably, these failures occurred on isolated data entries and were not reproducible in immediate re-execution, which further supports the hypothesis that they were transient rather than systematic issues.

VI. Opportunities for Future Work

Building on the findings discussed in **Error! Reference source not found.**, several opportunities emerge for extending the framework's capabilities and addressing current limitations. While the current implementation demonstrates a functional, modular, and scalable approach to GUI-based test automation in PowerOn, it also lays the foundation for a wide range of future enhancements. Similar initiatives in other utilities, such as UK Power Networks, National Grid and international operators in Australia and North America, have explored hybrid testing approaches combining GUI automation with backend validation. Incorporating lessons from these implementations could accelerate the evolution of the framework. These opportunities span both technical refinements and strategic extensions, aimed at increasing robustness, usability, intelligence, and integration with enterprise-level systems.

The key areas where this framework can evolve in future iterations are outlined in the following sections.

A. Improved Test orchestration and Execution Control

The current system is designed to execute tests sequentially, with batch support based on CSV-driven inputs. However, its architecture allows for:

- **Test queue management:** Introducing a visual or file-based interface where testers can assemble, reorder, and prioritise test cases.
- **Parameterized test templates:** Allowing dynamic substitution of values within test scripts (such as test location, test type or scale).
- **Custom test schedules:** Enabling timed or conditional execution based on system state, test importance, or resource availability.

These changes would move the tool from a script-based runner to a test orchestration platform, improving flexibility and collaborative workflows. Given the current batch execution's high success rate in the Network Diagram case study, improved orchestration could extend similar reliability to a broader range of scenarios.

B. Expansion of Test Case Library

While the initial development phase focused on navigation through the different modules and verifying correct operation of modules due to access restrictions (it is not possible to edit or change anything), the PowerOn platform contains a wide variety of functionalities that could also benefit from automated testing.

Extending the test case library into these areas would promote broader confidence in system integrity, uncover regression vulnerabilities, and support cross-module integration testing as PowerOn evolves. Such expansion would not only broaden test coverage but also serve as foundation for standardised regression suites within the RTS department.

C. Integration with Backend Verification Layers

At present, the framework is limited to assessing what is visually observable. However, PowerOn generates backend events that could be used to validate internal logic and consistency beyond the UI layer.

This would enable hybrid testing, combining the strengths of visual validation with programmatic verification, creating a more complete and reliable quality assurance framework. This hybrid model mirrors approaches adopted in other mission-critical systems, where front-end validation is reinforced by backend state verification to improve fault detection rates.

D. centralised Results Portal and Reporting Dashboard

Currently, results are saved in structured CSV logs, which are ideal for traceability and later processing. However, transforming this output into a real-time, interactive dashboard would greatly enhance its operational value.

This would include a centralised web portal for the RTS team, with visual dashboards highlighting key metrics, advanced search and filtering capabilities tools to allow engineers to explore results, and PDF or HTML reports that could be generated automatically.

Such capabilities would turn the framework from a standalone test tool into a continuous quality monitoring system, supporting audits, historical analysis, and management-level reporting. Beyond operational convenience, such dashboards could provide key performance indicators for management, aligning testing outcomes with organisational performance metrics.

E. Integration with CI/CD Pipelines

Although PowerOn is not natively integrated into CI/CD ecosystems, many of the test scripts and reporting mechanisms could be adapted to post-deployment validation scenarios.

This type of integration would align the RTS testing strategy with modern DevOps principles, promoting rapid delivery without sacrificing system reliability. Although full CI/CD integration may not be immediately achievable in PowerOn's production environment, establishing this capability in staging or training systems could pave the way for gradual adoption of DevOps-aligned practices.

VII. CONCLUSIONS

The development and execution of automated testing procedures for the PowerOn SCADA environment have demonstrated that a structured, image-recognition-based automation framework can reliably interact with complex operational interfaces and reproduce user workflows with minimal human intervention.

The batch execution served as a practical proof of concept for the scalability of the framework. By systematically running the same test case against a dataset of real substation names, the framework proved capable of handling repetitive tasks, logging detailed execution data, and recovering from failures without requiring a restart of the entire campaign. This approach not only reduces the manual workload but also increases the repeatability and consistency of test executions, which is critical for ensuring reliability in operational environments.

The decision to limit batch testing to a single, representative test case was driven by time constraints during the project timeline. Nevertheless, this single-case execution provided sufficient insight into how the framework could perform under batch conditions and offered a solid template for scaling the approach to other functional areas of the SCADA system. The success of this campaign indicates that the methodology can be applied to additional tests with only minor adaptations to the scripts and datasets.

One of the most significant outcomes of this work was the identification of image recognition sensitivity as both a strength and a limitation. While high-precision matching ensures that deviations from expected UI states are promptly detected, it also increases the likelihood of false negatives when the visual rendering of an element changes slightly due to background processes, resolution differences, or other environmental factors. This insight opens the door to future enhancements, such as implementing adaptive similarity thresholds or integrating additional context-aware validation methods to reduce unnecessary failure reports.

From a performance perspective, the test executions were stable, with consistent run times across the dataset and reliable mechanism for capturing and storing screenshots, error messages, and execution timestamps. This combination of functional reliability and detailed logging greatly improves post-execution analysis and root-cause investigation, making

the toolset not only an execution engine but also a diagnostic aid.

In conclusion, the project has delivered:

- A functional, adaptable automation framework for PowerOn SCADA environment.
- A validated methodology for batch execution with robust error handling and detailed result tracking.
- Insight into current limitations and clear paths for improvements.

The results, obtained from 200 independent executions of the Network Diagram workflow, strongly suggest that extending this framework to cover additional test cases will further enhance efficiency, accuracy, and coverage of SCADA system validations. With the foundations now established, future development should focus on broadening the scope of automated tests, refining recognition algorithms, and integrating automated reporting dashboards to support ongoing operational testing and quality assurance.

APPENDIX A: ALIGNMENT WITH UNITED NATIONS SUSTAINABLE DEVELOPMENT GOALS (SDGS)

This project directly and indirectly contributes to several of the United Nations Sustainable Development Goals (SDGs), particularly in the context of sustainable energy systems, innovation in industry, and operational resilience in critical infrastructure.

A. SDG 7: Affordable and Clean Energy

By enabling more reliable, repeatable, and efficient testing of SCADA/ADMS systems used in electrical distribution, the framework supports the stability and efficiency of power networks. Improved testing reduces the likelihood of operational failures and accelerates the deployment of upgrades, contributing to the provision of affordable, reliable, and sustainable electricity to end users.

B. SDG 9: Industry, Innovation and Infrastructure

The automation approach fosters innovation in an area traditionally reliant on manual procedures. The modular, scalable architecture provides a foundation for digital transformation in utility operations, supporting the development of resilient infrastructure and promoting sustainable industrial processes.

C. SDG 11: Sustainable Cities and Communities

Reliable power distribution is essential for the functioning of modern cities. By enhancing the robustness and efficiency of control system testing, this project indirectly improves the resilience of urban energy supply, reducing service interruptions and their impact on communities.

D. SDG 12: Responsible Consumption and Production

Automation reduces the resource intensity of testing activities, decreasing the need for prolonged operator involvement and minimising wasted time and energy. This leads to more responsible operational practices and more efficient use of human and technical resources.

E. SDG 13: Climate Action

While the project does not directly reduce greenhouse gas emissions, the improved reliability and efficiency in power network operations can enable better integration of renewable energy sources and reduce energy losses in the grid. This contributes indirectly to climate change mitigation efforts.

In summary, the project's main contribution to the SDGs lies in strengthening the reliability and sustainability of critical energy infrastructure through innovative, low-cost automation. By reducing human error, accelerating testing cycles, and facilitating large-scale validation, it supports the transition to smarter, more resilient and more sustainable power systems.

REFERENCES

[1] V. C. Gungor and F. C. Lambert, "A survey on communication networks for electric system automation," Computer Networks, 2006.

[2] IEEE Power & Energy Society, "IEEE Guide for SCADA and Automation Systems," IEEE Std, 2020.

[3] General Electric, "PowerOn Control," GE Vernova , 2018. [Online]. Available: https://www.gevernova.com/grid-solutions/sites/default/files/resources/products/brochures/uos/poweron_control.pdf.

[4] Autolt Consulting Ltd., "Autolt v3 Documentation," Autolt, 2022. [Online]. Available: https://www.autoitscript.com/autoit3/docs/.

[5] Keysight Tecnologies, "Eggplant FUnctional User GUide," Keysight, 2023. [Online]. Available: https://docs.eggplantsoftware.com/.

[6] Ranorex GmbH., "Ranorex Studio User Guide," Ranorex, [Online]. Available: https://support.ranorex.com/hc/en-us.

[7] SmartBear Software, "TestComplete Documentation," SMARTBEAR, 2023. [Online]. Available: https://support.smartbear.com/testcomplete/docs/.

[8] SikuliX Developers, "SikuliX Documentation," SikuliX, 2023. [Online]. Available: https://sikulix.github.io/.

[9] W. T. Wang and X. Bai, "A GUI regression testing method based on dynamic event flow analysis," Third International Conference on Software Testing, Verfication and Validation Workshops, 2010.

[10] B. P. Lamancha and M. P. Usaola, "Automated testing in a GUI based application," Software Quality Journal, 2010.

[11] IEEE Standards Association, "IEEE Standard for Software and System Test Documentation," IEEE Std, 2013.

[12] G. J. Myers, C. Sandler and T. Badgett, "The Art of Software Testing," Wiley, 2011.