



MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE GRADO 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

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título
Design and implementation of a test automation strategy for PowerOn control systems
en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el
curso académico 2024/25 es de mi autoría, original e inédito y
no ha sido presentado con anterioridad a otros efectos.
El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido
tomada de otros documentos está debidamente referenciada.

Fdo.: Alberto López-Rey Rojas

Fecha: 12/08/2025

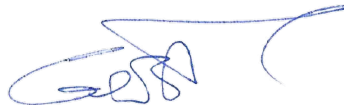


Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Graeme Burt

Fecha: 18/08/25.





MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE GRADO 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

Autor: López-Rey Rojas, Alberto.

Director: Burt, Graeme.

Entidad Colaboradora: Scottish Power

RESUMEN DEL PROYECTO

Este trabajo presenta el diseño e implementación de un marco de automatización de pruebas visuales para *PowerOn*, la plataforma SCADA/ADMS utilizada por Scottish Power, que permite superar la falta de acceso al backend y la ausencia de APIs. La solución, desarrollada con SikuliX, automatiza pruebas funcionales y de regresión mediante reconocimiento de imágenes, OCR e interacción simulada de usuario. En pruebas a gran escala sobre la interfaz de Diagrama de Red, el sistema alcanzó una tasa de éxito del 94% en 200 subestaciones, demostrando una elevada robustez, repetibilidad y escalabilidad.

Palabras clave: PowerOn, SCADA, ADMS, Automatización basada en GUI, Marco de Automatización de Pruebas, SikuliX, Reconocimiento de Imágenes, Reconocimiento Óptico de Caracteres (OCR), Pruebas por Lotes, Pruebas en Infraestructuras Críticas.

1. Introducción y Objetivos

En el sector eléctrico moderno, los sistemas SCADA (*Supervisory Control and Data Acquisition*) y ADMS (*Advanced Distribution Management Systems*) son esenciales para garantizar la fiabilidad de la red, optimizar la operación y cumplir con las normativas. *PowerOn*, desarrollado por General Electric Grid Solutions y utilizado por Scottish Power, es una plataforma SCADA/ADMS de misión crítica que integra diversos módulos operativos, como Diagrama de Red, GeoView, Gestor de Paquetes de Trabajo y sistemas de restauración de fallos.

Las pruebas en *PowerOn* se han realizado tradicionalmente de forma manual, mediante procedimientos extensos que requieren operadores expertos siguiendo guiones detallados y verificando visualmente los resultados. La falta de acceso al backend, de APIs públicas y de árboles de objetos inspeccionables imposibilita el uso de herramientas de automatización convencionales.

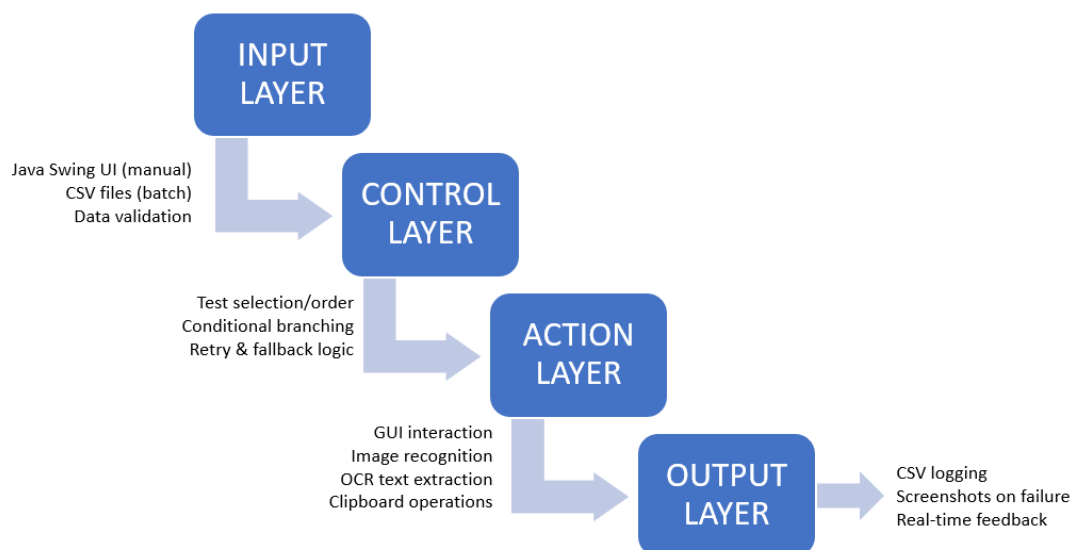
Este proyecto tuvo como objetivo diseñar e implementar un marco de automatización de pruebas visuales que interactúe exclusivamente con la interfaz gráfica de *PowerOn*, replicando las acciones de un operador humano y permitiendo la ejecución de pruebas funcionales y de regresión a gran escala. Los objetivos específicos son:

- Desarrollar una arquitectura modular y extensible compatible con entornos de solo interfaz gráfica.
- Implementar casos de prueba representativos de los flujos de trabajo del departamento de Sistemas en Tiempo Real (RTS).
- Validar la escalabilidad mediante la ejecución por lotes sobre grandes volúmenes de datos.

2. Metodología

El marco se desarrolló con SikuliX 2.0.5, herramienta de código abierto que emplea reconocimiento de imágenes basado en OpenCV y OCR con Tesseract para identificar elementos en pantalla y simular interacciones de teclado y ratón. La arquitectura se organizó en cuatro capas, como se muestra en la imagen:

1. **Entrada:** parámetros introducidos desde una interfaz Java Swing (modo manual) o a partir de ficheros CSV (modo por lotes).
2. **Control:** gestión del flujo de pruebas, bifurcaciones condicionales, reintentos y lógica de recuperación.
3. **Acción:** ejecución de interacciones con la interfaz (clics, escritura, desplazamiento) y validaciones mediante OCR.
4. **Salida:** generación de registros estructurados con resultados, causas de error, marcas de tiempo y capturas de pantalla.



La gestión de imágenes se basó en:

- Bibliotecas organizadas por interfaz y función
- Variantes múltiples por elemento para detección de respaldo.
- Umbrales de similitud adaptativos (0,3–0,99) para equilibrar precisión y tolerancia a cambios visuales.
- Validación secuencial de estados con `wait()` y `waitVanish()` para asegurar la disponibilidad de la interfaz antes de actuar.

La ejecución por lotes se implementó mediante control por CSV, lo que permitió aplicar un mismo procedimiento a cientos de subestaciones sin modificar el código. Además, se incorporó lógica de “watchdog” para cerrar automáticamente ventanas emergentes inesperadas y evitar interrupciones

3. Resultados y Discusión

La validación se realizó en el entorno de pruebas de *PowerOn* (v6.9.3) con un caso de estudio centrado en la localización de subestaciones y su validación visual en distintos módulos.

Resultados principales:

- Tasa de éxito del 94% en 200 subestaciones en la interfaz de Diagrama de Red.
- Los fallos detectados fueron puntuales, originados por pequeños retrasos de renderizado, y se resolvieron con la lógica de reintento y el sistema “watchdog”.
- La ejecución por lotes generó registros CSV completos con marcas de tiempo, descripciones de errores y capturas de pantalla de los casos fallidos.

La siguiente imagen presenta los resultados de la ejecución por lotes, mostrando si se ha completado o no (pass/fail), y los errores que se dan en caso negativo.

Substation	Result	Error	Failed Step	Screenshot	Durati	Timestamp
AYR GSP	Pass				15.432	10/08/2025 11:18
BONNYBRIDGE GSP	Pass				9.049	10/08/2025 11:18
CARINTYNE 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 NEW	Pass				9.915	10/08/2025 11:19
CROOKSTON 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				10.321	10/08/2025 11:20
DUMFRIES GSP	Pass				10.181	10/08/2025 11:20
DUMFERMLINE GSP	Pass				10.531	10/08/2025 11:20
EASTERHOUSE GSP	Pass				9.243	10/08/2025 11:20
ELDELSIE 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: 175193205482	Open scale	C:\Users\B620717\Desktop\PowerOnResults\snap_auto_fail_20250810_112116.png	13.1	10/08/2025 11:21
GLENLUCE GSP	Fail	[Open filters] Cannot click; not found: 17520646957	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				9.024	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 GSP	Pass				9.176	10/08/2025 11:22
KILBOWIE GSP	Pass				8.867	10/08/2025 11:22
KILLERMONT GSP NEW	Pass				9.25	10/08/2025 11:22
KILMARNOCK SOUTH GSP	Pass				9.355	10/08/2025 11:23
KILWINNING GSP	Pass				9.259	10/08/2025 11:23
LINNEMILL GSP	Pass				9.212	10/08/2025 11:23
LIVINGSTON EAST GSP	Pass				9.287	10/08/2025 11:23
MAYBOLE GSP	Pass				8.395	10/08/2025 11:23
NEARTHILL GSP (NEW)	Pass				9.214	10/08/2025 11:23
NEWTON STEWART 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 ANDREW'S CROSS GSP	Pass				8.464	10/08/2025 11:24
STIRLING GSP	Pass				8.981	10/08/2025 11:24
STRATHAVEN GSP	Pass				9.162	10/08/2025 11:24
STRATHLEVEN GSP	Pass				9.175	10/08/2025 11:25
TONGLAND GSP (T)	Pass				9.917	10/08/2025 11:25
WEST GEORGE ST GSP	Pass				9.962	10/08/2025 11:25
WESTBURN ROAD GSP	Pass				9.639	10/08/2025 11:25
WETBURN ROAD GSP	Pass				9.594	10/08/2025 11:25
WHISTLEFIELD GSP	Pass				9.661	10/08/2025 11:25
WISHAW GSP	Pass				9.43	10/08/2025 11:26
WISHAW GSP A	Pass				8.579	10/08/2025 11:26
WISHAW GSP B	Pass				9.458	10/08/2025 11:26

En comparación con las pruebas manuales, el marco demostró:

- **Mayor velocidad:** hasta cinco veces más rápido.
- **Mayor repetibilidad:** eliminación de la variabilidad humana en la ejecución y verificación.
- **Mayor escalabilidad:** capacidad de ejecutar pruebas masivas de forma desatendida durante la noche.

Aunque se evaluó la herramienta *Eggplant*, con capacidades técnicas superiores, su coste de licencia (aprox. 10.000 €) la hizo inviable. SikuliX ofreció prestaciones suficientes para alcanzar los objetivos y una base sólida para futuras ampliaciones.

4. Conclusiones

Este trabajo demuestra que la automatización basada en interfaz gráfica es una estrategia eficaz para entornos SCADA/ADMS cerrados como *PowerOn*. Con herramientas de código abierto fue posible:

- Ejecutar flujos de trabajo visualmente complejos sin acceso al backend, con alta tasa de éxito.
- Diseñar una arquitectura escalable para pruebas tanto manuales como por lotes.
- Proporcionar evidencias estructuradas y auditables para uso operativo y regulatorio.

Como líneas futuras se propone:

- Ampliar la cobertura de pruebas a más módulos y escenarios de *PowerOn*.
- Integrar verificaciones de backend para una validación híbrida.
- Incorporar un panel centralizado de informes y enlazar el marco a pipelines CI/CD para pruebas continuas.

En resumen, el marco desarrollado constituye una solución de bajo coste, adaptable y robusta para la automatización de pruebas en sistemas críticos donde los métodos tradicionales no son aplicables.

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

Author: López-Rey Rojas, Alberto.

Supervisor: Burt, Graeme.

Collaborating Entity: Scottish Power

ABSTRACT

This work presents the design and implementation of a visual test automation framework for PowerOn, the SCADA/ADMS platform used by Scottish Power, overcoming the absence of backend access and APIs. The solution developed with SikuliX, automates functional and regression testing through image recognition, OCR, and simulated user interaction. In large-scale tests on the Network Diagram interface, the framework achieved a 94% success rate cross 200 substations, demonstrating high robustness, repeatability, and scalability.

Keywords: PowerOn, SCADA, ADMS, GUI-based Automation, Test Automation Framework, SikuliX, Image Recognition, Optical Character Recognition (OCR), Batch Testing, Critical Infrastructure Testing.

1. Introduction and Objectives

Modern power utilities rely on Supervisory Control and Data Acquisition (SCADA) and Advanced Distribution Management Systems (ADMS) to ensure network reliability, operational efficiency, and compliance with regulatory requirements. PowerOn, developed by General Electric Grid Solutions and used by Scottish Power is a mission-critical SCADA/ADMS platform with multiple operational modules such as Network Diagram, GeoView, Work Package Manager, and fault restoration systems.

Testing in PowerOn is traditionally a manual, time-consuming process involving expert operators following lengthy scripts and visually confirming outcomes. The absence of backend access, public APIs, or inspectable object trees makes conventional automation tools unsuitable.

This project aimed to design and implement a visual test automation framework that interacts exclusively with PowerOn's graphical interface, replicating human operator actions to enable functional and regression testing at scale. The goals were to:

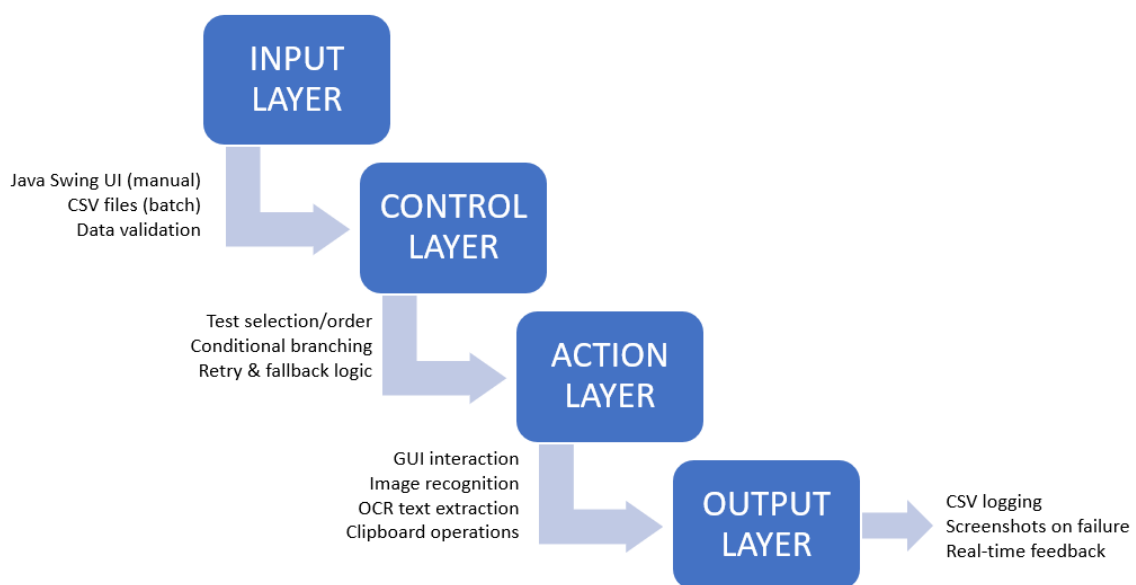
- Develop a modular, extensible automation architecture compatible with GUI-only environments.
- Implement representative test cases aligned with existing Real-Time Systems (RTS) department workflows.
- Validate scalability through batch execution over large datasets.

2. Methodology

The automation framework was implemented using SikuliX 2.0.5, an open-source tool leveraging OpenCV-based image recognition and Tesseract OCR to detect on-screen elements and simulate keyboard and mouse inputs. The solution was designed in four layers:

1. **Input Layer:** Accepts parameters via Java Swing GUI (manual mode) or CSV datasets (batch mode).
2. **Control Layer:** Orchestrates test flow, conditional branching, retries, and fallback logic.
3. **Action Layer:** Executes GUI interactions (clicks, typing, scrolling) and performs OCR validations.
4. **Output Layer:** Logs structured results (Pass/Fail/Partial), error causes, timestamps, and screenshots into CSV files.

Following figure shows the high-level architecture of the framework, highlighting modularity, data-driven execution, and error handling mechanisms.



The image-handling strategy included:

- **Structured image libraries** grouped by interface and function.
- **Fallback detection** with multiple image variants per elements.
- **Adaptive similarity thresholds** (from .3 to 0.99) to balance accuracy and tolerance to UI changes.
- **Sequential state validation** using combined *wait()* and *waitVanish()* calls to ensure readiness before actions.

Batch testing capabilities were enabled by CSV-driven execution, allowing the same procedure to be run across hundreds of substations without code changes. Watchdog logic was incorporated to detect and dismiss unexpected popups, ensuring uninterrupted runs.

3. Results and Discussion

The framework was validated in the PowerOn test environment (v6.9.3) through a representative case study automating substation location and visual validation in several modules.

Key performance results:

- 94% success rate over 200 substations in the Network Diagram interface.
- Failures were transient and caused mainly by minor rendering delays, successfully mitigated by retry and watchdog logic.
- Batch execution produced complete CSV logs with timestamps, error descriptions, and screenshots for failed cases.

The following figure presents the aggregated batch execution results, showing pass/fail distribution and error categories.

Substation	Result	Error	Failed Step	Screenshot	Duration	Timestamp
Ayr GSP	Pass				15.432	10/08/2025 11:18
BONNYBRIDGE GSP	Pass				9.049	10/08/2025 11:18
CARINTYNE 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 NEW	Pass				9.915	10/08/2025 11:19
CROOKSTON 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				10.321	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: 175199205482	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: 17520646957	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				9.024	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 GSP	Pass				9.176	10/08/2025 11:22
KILBOVIE GSP	Pass				8.667	10/08/2025 11:22
KILLERMONT GSP NEW	Pass				9.25	10/08/2025 11:22
KILMARNOCK SOUTH GSP	Pass				9.355	10/08/2025 11:23
KILWINNING GSP	Pass				9.259	10/08/2025 11:23
LINN MILL GSP	Pass				9.212	10/08/2025 11:23
LIVINGSTON EAST GSP	Pass				9.287	10/08/2025 11:23
MAYBOLE GSP	Pass				8.395	10/08/2025 11:23
NEWARTHILL GSP (NEW)	Pass				9.214	10/08/2025 11:23
NEWTON STEWART 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
SALT COATS B	Pass				9.21	10/08/2025 11:24
ST ANDREW'S CROSS GSP	Pass				8.464	10/08/2025 11:24
STIRLING GSP	Pass				8.981	10/08/2025 11:24
STRATHAVEN GSP	Pass				9.162	10/08/2025 11:24
STRATHLEVEN GSP	Pass				9.175	10/08/2025 11:25
TONGLAND GSP (T)	Pass				9.917	10/08/2025 11:25
WEST GEORGE ST GSP	Pass				9.962	10/08/2025 11:25
WESTBURN ROAD GSP	Pass				9.639	10/08/2025 11:25
WETBURN ROAD GSP	Pass				9.594	10/08/2025 11:25
WHISTLEFIELD GSP	Pass				9.661	10/08/2025 11:25
WISHAW GSP	Pass				9.43	10/08/2025 11:26
WISHAW GSP A	Pass				8.579	10/08/2025 11:26
WISHAW GSP B	Pass				9.458	10/08/2025 11:26

Comparison with manual testing indicated improvements in:

- **Speed:** automated runs executed up to 5 times faster than manual equivalents.
- **Repeatability:** elimination of human variability in execution order and verification criteria.
- **Scalability:** ability to run unattended, large-scale regression tests overnight.

While Eggplant was identified as the technically superior visual automation tool, its high licensing cost (€10,000 approximately) made it unsuitable for the project's budget constraints. SikuliX provided sufficient functionality to meet the objectives and delivered a proof-of-concept ready for future expansion.

4. Conclusions

This project demonstrated that GUI-based automation is a viable and effective strategy for testing closed SCADA/ADMS environments like PowerOn. By leveraging open-source tools, it was possible to:

- Achieve high success rates in visually complex workflows without backend access.
- Implement a scalable architecture supporting both manual-triggered and fully automated batch runs.
- Provide structured auditable test evidence suitable for operational and compliance needs.

Future work should focus on:

- Expanding the test library to cover more PowerOn modules and operational scenarios.
- Integrating backend verification layers for hybrid visual-data validation.
- Developing a centralized reporting dashboard and connecting the framework to CI/CD pipelines for continuous deployment testing.

In conclusion, the proposed framework establishes a low-cost, adaptable, and robust foundation for test automation in critical infrastructure systems where traditional methods cannot be applied.

Contents

1	Introduction.....	1
1.1	Project Scope and Objectives.....	2
2	State of the Art and Background for Test Automation in Closed Systems	5
2.1	SCADA Testing Context.....	5
2.2	State of the Art in Test Automation for Closed Systems.....	6
2.3	International Context and Industry Landscape	8
2.4	Tool Evaluation and Comparison	8
2.4.1	SikuliX.....	9
2.4.2	Eggplant	9
2.4.3	Autolt	10
2.4.4	Ranorex.....	10
2.4.5	TestComplete	10
2.4.6	Comparative Tool Analysis	11
2.4.7	Conclusion of Tool Evaluation.....	12
3	Design and Methodology	13
3.1	Environment Setup and Tools Used	13
3.1.1	PowerOn Access and Restrictions	13
3.1.2	Tools and Libraries	13
3.1.3	Development Environment.....	16
3.2	Test Automation Architecture	16
3.2.1	Structure and Modularity	17
3.2.2	Image-Based Logic and Handling.....	19
3.2.3	Scalability for Batch Testing	24

3.2.4	Error Handling and Watchdog Logic	26
4	Case Study	29
4.1	Workflow Example: End-to-End Test Case	29
4.1.1	Step-by-Step Description	29
4.2	Results Logging and Reporting.....	34
4.2.1	Structure of the Result Log	34
4.2.2	Execution Feedback During Runtime.....	36
4.2.3	Support for Batch Aggregation	38
4.2.4	Design for Auditability and Expansion.....	39
4.3	Limitations and Considerations.....	40
4.3.1	Dependence on Visual Environment	40
4.3.2	Image Maintenance Overhead	40
4.3.3	Limited to Observable Behaviour	41
4.3.4	Performance Variability and System Load	41
4.3.5	Manual Preconditions and Operator Involvement.....	42
4.3.6	Lack of Advanced AI Integration	42
5	Results Analysis	44
5.1	Execution Summary	44
5.2	Success Rate Interpretation	45
5.3	Failure Analysis.....	45
5.4	Observations on Robustness and Error Handling	46
5.5	Implications for Future Testing	46
6	Opportunities for Future Work	47
6.1	Improved Test Orchestration and Execution Control.....	47
6.2	Expansion of Test Case Library	47

6.3	Integration with Backend Verification Layers	48
6.4	Centralised Results Portal and Reporting Dashboard	48
6.5	Integration with CI/CD Pipelines	49
7	Conclusions.....	50
8	References	52
9	Appendix A: Single Tests Code	54
10	Appendix B: Batch Tests Code	75

1 Introduction

In the current landscape of power distribution management, the demand for robust, efficient, and resilient control systems is greater than ever. Power utilities must ensure continuous service availability [1] [2], fast fault response, and real-time visibility over increasingly complex networks; all while meeting regulatory requirements and pursuing digital transformation goals. At the core of this digital infrastructure are Supervisory Control and Data Acquisition (SCADA) and Distribution Management Systems (DMS), which enable operators to monitor and control grid components, respond to faults, and optimise operations. One such system is PowerOn, developed by General Electric Grid Solution [3], which plays a central role in the operation and supervision of the electrical distribution network at companies like Scottish Power.

PowerOn integrates a wide range of modules, including interactive network diagrams, geographical mapping interfaces, automated fault response systems such as APRS and PORT, and a variety of network monitoring, simulation, and planning tools. Operators rely on it to locate substations, open or close switches, analyse load flow, restore service after faults, and execute daily operational tasks. However, despite its powerful capabilities, the system presents unique challenges when it comes to testing, validating, and updating its software environment.

Testing new features or configurations in PowerOn is currently a highly manual process, dependent on expert operators following long test scripts and visually confirming results. This not only increases the chance of human error but also limits the scalability and repeatability of test cycles. Furthermore, because PowerOn does not expose a public API or backend access for automation, conventional automated testing tools; such as Selenium, Cypress or even Python-based frameworks are not compatibles. These limitations necessitate the exploration of alternative testing strategies that can interact with the system at the visual level, much like a human operator would.

This MSc project seeks to address that gap by developing a visual test automation framework for PowerOn using SikuliX, an open-source tool that leverages image recognition and scripting to automate GUI interactions. The framework aims to simulate a variety of testing scenarios (including smoke tests, regression checks, exploratory system trials, and interface validation) by detecting visual components, simulating mouse and keyboard inputs, and validating system responses using OCR (Optical Character Recognition). By doing so, it provides a foundation for repeatable and time-efficient testing that can be used both during development cycles and post-deployment updates.

The approach has been shaped by key constraints of the PowerOn environment. First, access to the system is limited to graphical interface, and user login must be performed manually due to security policies. Second, many screen elements change position or appearance depending on system state, which requires flexible image recognition techniques. Third, some operations (like scaling diagrams or navigating nested menus) are difficult to automate precisely without introducing human-like logic and checks. These limitations make SikuliX a particularly suitable tool, as it does not depend on internal access or pre-built automation hooks, it works entirely by visually identifying and interacting with screen content.

Throughout the project, development has been guided by regular collaboration with the Real-Time Systems (RTS) department at Scottish Power, who provided access to the testing environment and continuous feedback on the implementation. Weekly meetings with industrial and academic

supervisors ensured that the work aligned with both technical needs and academic expectation. Early stages of the project involved reviewing existing manual testing procedure, understanding RTS workflows, and evaluating previous automation attempts. Later stages focused on developing reusable SikuliX functions to interact with specific parts of PowerOn, such as toolbar control, substation search, diagram navigation, scale adjustment, or interface validation.

The framework was designed with portability and flexibility in mind. For instance, all user input (e.g., substation names) can be loaded dynamically from external CSV or Excel files, and image references are modular to allow easy substitution if the interface design changes. Test can be selected dynamically by the user at runtime using visual checkboxes. Interaction timing is synchronised with the system state by the use of commands like *exists()* or *waitVanish()* to ensure robustness. Where possible, the system relies on clipboard-based text input (instead of simulated typing) to increase speed and reduce the human-like appearance of automation.

Although the project duration was limited to a three-month placement, significant progress was made in demonstrating the feasibility of visual test automation in closed, graphical systems like PowerOn. Several core test flows were successfully automated, including substation location via both Network Diagram and GeoView, interface response validation, and toolbar interaction. A library of modular functions was created to support future extension.

Additionally, to support both academic analysis and business impact, the final framework is designed to run automated tests across a large batch of substations and collect the outcomes systematically. The results of these test runs, whether successful, partially complete, or failed; are recorded into a structured CSV report. This allows for quantitative performance evaluation, debugging or inconsistent behaviours, and supports the generation of audit logs. This output mechanism enhances the project's value for the RTS department, offering traceable insights into system behaviour during updates or changes.

In addition, the approach was benchmarked against traditional manual processes, showing potential gains in speed, repeatability, and accuracy.

This report documents the complete project journey, from background research and tool selection to implementation, evaluation, and final conclusions. It also discusses the implications of this approach for other SCADA environments with similar constraints, and offers recommendations for future development such as full CI/CD integration, hybrid visual + AI testing models, and potential extensions beyond PowerOn. Ultimately the project contributes a proof-of-concept methodology for non-intrusive test automation in critical infrastructure, enabling more reliable, scalable, and modern testing practices in environments where traditional automation cannot be applied.

1.1 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.

2 State of the Art and Background for Test Automation in Closed Systems

2.1 SCADA Testing Context

Modern electrical distribution systems rely heavily on advanced digital infrastructures to ensure reliable, real-time monitoring and control of the network. Among these, SCADA (Supervisory Control and Data Acquisition) systems are fundamental. They allow operators to remotely supervise substations, monitor loads, detect faults, and coordinate the restoration of service. While SCADA systems provide an essential backbone for operational continuity [1], their increasing complexity has made testing, maintenance and feature development significantly more challenging.

PowerOn, the SCADA/ADMS platform developed by General Electric Grid Solutions, is one such system used by Scottish Power to operate its distribution and transmission networks. It offers various modules, including real-time network diagrams, geographical mapping through GeoView, automatic restoration via APRS and PORT, and tools for job planning and fault simulation. However, PowerOn is a closed system (its source code is inaccessible, and no public APIs are exposed for automation or integration), and its graphical interface must be navigated manually. These constraints limit the types of test automation that can be applied as conventional tools cannot be used, as they rely on code-level hooks or HTML structures. Therefore, automation within PowerOn requires a tool capable of interacting visually with the GUI, mimicking human behaviour.

Figure 1 provides a visual overview of the PowerOn environment. Each subfigure focuses on a specific interface relevant to the testing framework: Network Diagram, System Overview, Work Package Manager, and GeoView, respectively.

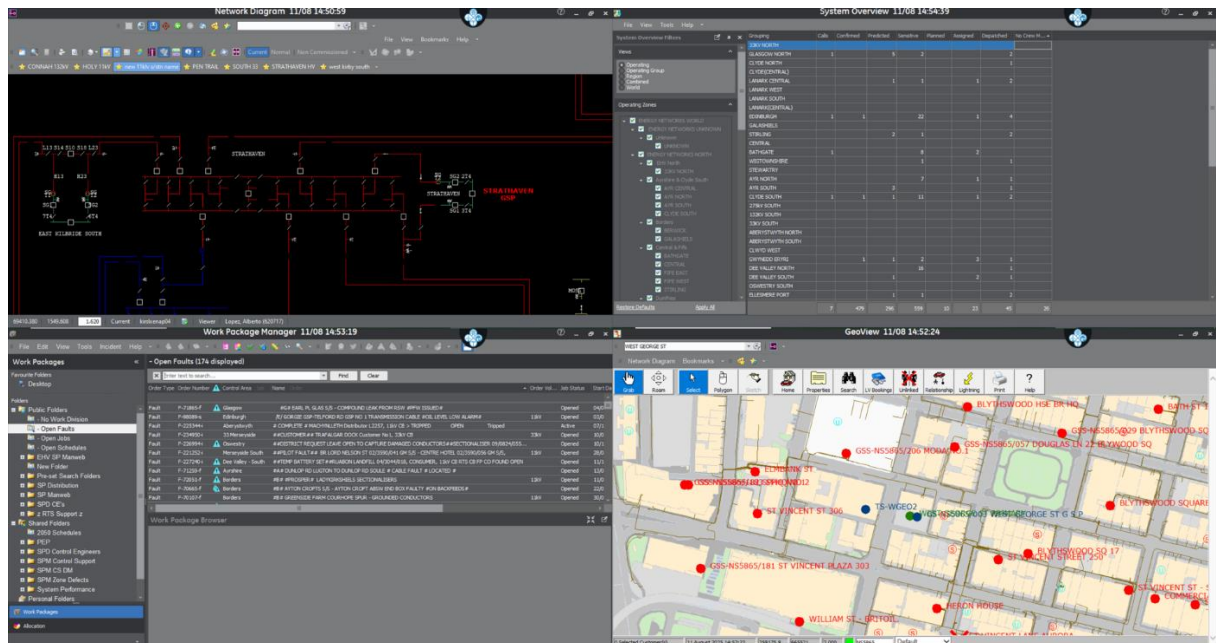


Figure 1. PowerOn module interfaces. Include Network Diagram (up left), System Overview (up right), Work Package Manager (down left), and GeoView (down right)

2.2 State of the Art in Test Automation for Closed Systems

While test automation has seen wide adoption in mainstream software engineering, especially for web and mobile applications, the automation of complex, closed, or GUI-only systems remains a much more constrained field [4] [5]. This is especially true for environments that do not expose backend access, API endpoints or test hooks. Examples include legacy enterprise software, industrial control systems (like SCADA), financial tools, and medical platforms.

For instance, banking systems and electronic medical records platforms are often proprietary and security-restricted. Testing these systems requires navigating graphical interfaces built decades ago, with no support for modern testing frameworks like Selenium or Cypress. In these cases, organisations have increasingly adopted visual automation tools, solutions that replicate human interaction by analysing what is on screen, identifying key UI elements through image recognition, and simulating inputs via mouse and keyboard events.

One leading tool in this space is Eggplant (owned by Keysight). It has been adopted in critical sectors such as:

- NHS Digital (UK): for automated testing of medical systems, reducing the time needed to validate complex workflows involving patient data, UI safety, and access control.
- MCB Bank (Pakistan): for automating testing of internal finance platforms that lacked API access but required reliable regression testing under compliance standards.
- UK Power Networks: reportedly exploring model-based and visual test methods for parts of its control systems.

Eggplant uses a model-based approach, combined with AI and OCR, to create adaptable tests that reflect user journeys. However, it is a commercial product with substantial licensing fees, and it depends on its own scripting language (SenseTalk), which can slow adoption and reduce developer flexibility. [6]

Another tool explored is Autolt, which is a Windows-focused scripting language designed for automating user interactions in desktop environments. It excels at simulating keyboard and mouse inputs interact with system-level applications. Originally built for automating routing tasks in Windows, Autolt provides an intuitive syntax and lightweight execution, making it popular among IT support and QA teams in small-scale environments.

However, its capabilities are fundamentally limited to input simulation and basic window control. Unlike tools such as SikuliX or Eggplant, Autolt does not include built-in visual recognition or OCR functionality. As such, it cannot reliably identify or interact with dynamic or graphical UI elements which makes it less suitable for visually rich systems like PowerOn. In environments where element positions, icons, or screen layout vary, Autolt scripts are prone to failure or misalignment. Nevertheless, its ease of use and native integration with Windows make it a valuable tool for automating stable, predictable GUIs.

Ranorex is another commercial test automation framework built for desktop, web, and mobile applications. It leverages a UI object recognition engine that allows testers to identify and interact with on-screen elements using a proprietary automation API. Ranorex includes features such as test recording, test management, and reporting dashboards. It supports popular development environments like .NET and provides strong support for hybrid applications with complex UI structures.

However, Ranorex relies heavily on access to an internal object tree or UI hierarchy, something that is simply not available in closed platforms like PowerOn. In these scenarios, where all interaction must occur through image recognition and external control, Ranorex loses much of its functionality. This technical dependency limits its applicability in highly restricted environments or when internal access is not feasible due to licensing, security, or architecture constraints.

Similar to Ranorex, TestComplete is a powerful commercial automation tool developed by SmartBear. It supports a wide range of applications and is known for its scriptless test creation interface and advanced test analytics. TestComplete enables users to write tests in multiple scripting languages (JavaScript, Python, VBScript) and integrates well with CI/CD pipelines and test management platforms.

The key limitation is that TestComplete requires access to internal object properties. It uses object-based locators to interact with the UI, and without the ability to introspect the application structure (object IDs, class names), it cannot function properly. In the context of PowerOn, where the UI is presented only as pixels on the screen with no programmatic accessibility, TestComplete becomes unusable without extensive workarounds.

SikuliX, on the other hand, offers a fundamentally different approach by focusing exclusively on image-based visual automation. Built on top of OpenCV and Tesseract OCR, SikuliX allows test scripts to identify and interact with screen elements based purely on visual patterns [7] [8]. This makes it uniquely suitable for applications like PowerOn, where access is restricted to the graphical interface and there is no programmatic way to inspect or manipulate UI elements directly.

What distinguishes SikuliX is its simplicity and openness, it is fully open-source, scriptable in Jython (Python for the JVM), and supports seamless integration with Java Swing for lightweight UI components. OCR capabilities allow it to read screen text when necessary [9]. Additionally, SikuliX is small in size, platform-independent, and does not require complex installation or licensing.

Across all these contexts, a recurring theme emerges: whenever systems cannot be accessed via APIs or structured identifiers, automation must revert to visual simulation (essentially emulating what a human user would do). This makes visual automation not only a viable fallback strategy, but in many cases the only feasible solution for end-to-end test coverage.

Furthermore, the rise of AI-enhanced testing is also impacting this space. Some platforms now explore ways of auto-generating test scenarios, auto-detecting unexpected UI changes, or simulating exploratory testing logic using AI. While promising, these technologies still rely on a foundation of repeatable visual control.

By situating this project within this wider context, not just SCADA but also banking, healthcare, and legacy IT; it becomes clear that this work addresses a broader class of automation problems. It provides a scalable, low-cost, and adaptable alternative to commercial tools, and demonstrates that even

restricted environments like PowerOn can benefit from structured, repeatable, and intelligent test automation through visual means. [10] [11]

2.3 International Context and Industry Landscape

Utilities and vendors worldwide are actively exploring approaches to automate testing in closed, GUI-only operational environments (such as SCADA/ADMS), often combining visual GUI automation with complementary techniques. Although detailed utility test assets are rarely public due to security constraints, published case studies and industrial research indicate a clear trend toward automated, image/OCR-driven testing and hybrid validation strategies.

On the utility side, public material points to large operators modernising their test automation stacks with visual, model-based tools. For example, American Electric Power (AEP) is referenced by Keysight/Eggplant in an enterprise test-automation modernisation case study highlighting Eggplant's role in improving automation and DevOps outcomes, evidence of adoption within a major North American utility context. [12]

From the research and industrial practice perspective, an extended industrial study on visual GUI testing documents how image-based techniques are applied in practice, and well-cited SCADA software case study reports automated testing of a commercial, large-scale SCADA system, both supporting the feasibility of automation in critical, closed systems.

On the vendor landscape, mature platforms exist that support GUI-centric automation relevant to closed environments (e.g., computer-vision/OCR-driven or model-based tools). A concise comparison of leading options and their fit to PowerOn is provided in Tool Evaluation and Comparison. In short, visual and model-based approaches are frequently adopted where UIs are "pixel-only", while object-centric tools (such as Ranorex, TestComplete) excel when an accessible UI hierarchy is available. Enterprise-scale MBT suites dominate end-to-end process testing and can be paired with image/OCR capabilities in closed context.

Combined with the evidence from utilities and industrial studies, the international trend points to hybrid validation (visual GUI automation + backend/log checks) as the pragmatic path to robustness in closed systems, the same direction proposed for extending our framework.

2.4 Tool Evaluation and Comparison

One of the early stages of the project involves conducting a comprehensive evaluation of available automation tools to determine their suitability for automating PowerOn. Given the limitations of the system (absence of an API, restricted backend access, and reliance on GUI interaction) tools that depend on code introspection or DOM-level interaction were immediately excluded.

Instead, the focus shifted to tools capable of interacting with the graphical user interface in a visual manner, replicating the actions of a human operator. Several candidates were shortlisted and reviewed in depth: SikuliX, Eggplant, Autolt, Ranorex, and TestComplete. Each tool was studied considering factors such as ease of integration, robustness, visual recognition accuracy, and overall flexibility.

2.4.1 SikuliX

SikuliX is an open-source GUI automation tool that uses image recognition to identify and interact with on-screen elements. It is written in Java and supports scripting through Jython. SikuliX operates entirely at the screen level using screenshots to simulate clicks, keystrokes, and visual verifications. This allows it to function in environments that are entirely closed or graphical in nature. [7]

Its main positive aspect can be summarized as follows:

- Completely open-source and free.
- Excellent for automating GUI-only environments.
- Allows clipboard interaction, OCR, and screen monitoring.
- Easy to prototype and extend using Python-like syntax
- Works well in low-permission environments.

On the other hand, it presents the following challenges:

- OCR quality can vary depending on resolution and font.
- It is sensitive to screen layout changes (requires robust image matching).
- Lacks built-in reporting or test management.

2.4.2 Eggplant

Eggplant is a commercial visual automation platform that provides advanced AI-driven model-based testing. It supports image recognition, OCR, and behavioural test scripting. Eggplant is widely used in sectors such as healthcare (NHS Digital) or banking (Mauritius Commercial Bank) for automating complex systems that are difficult to access or integrate. [6]

Among its main strengths are the following:

- AI-based test generation and adaptive test models.
- High-accuracy OCR and robust image matching.
- Integrated dashboards and analytics.
- Used successfully in other closed-systems industries.

Despite these advantages, some drawbacks include:

- High licensing cost (10,000 €) and proprietary technology (SenseTalk).
- Requires training and investment in infrastructure.
- Overkill for lightweight scripting or quick deployments.

2.4.3 Autolt

Autolt is a lightweight scripting language designed for automating Windows-based tasks. It simulates mouse movements, clicks, keystrokes, and window manipulation. It is best suited for automating stable desktop applications or administrative tasks. [13]

This tool offers several notable advantages:

- Extremely lightweight and fast.
- Easy to learn for basic automation.
- Good for static GUIs or legacy apps.

On the other hand, it also has certain limitations:

- No OCR or image matching capabilities.
- Not suitable for dynamic or graphical systems.
- Limited error handling and test structure.

2.4.4 Ranorex

Ranorex is a commercial test automation framework that targets desktop, web, and mobile applications. It provides UI element inspection, test recording, and integrates with test management platforms. [14]

Some of the key benefits include:

- Advanced UI object recognition.
- Includes test recorder and structured test design.
- Well-supported and scalable.

On the other hand, there are a few notable weaknesses like:

- Relies on object tree access.
- Expensive commercial license.
- Unusable in systems that present only visual output.

2.4.5 TestComplete

TestComplete is another commercial tool widely used in enterprise automation. It supports scriptless testing, OCR, and multi-platform compatibility. It is designed to integrate easily into CI/CD pipelines and comes with strong analytics tools. [15]

Its strengths are the following:

- Multi-language scripting support.

- Integrated reporting and dashboards.
- Flexible test design options.

However, it has the following limitations:

- Requires UI object inspection.
- Complex setup and learning curve.
- Commercial licensing cost.

2.4.6 Comparative Tool Analysis

Table 1 provides a consolidated overview of the key strengths and weaknesses of each automation tool evaluated during this project. The scoring is based on both technical documentation 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.

Tools like Autolt, Ranorex, and TestComplete fall short in this context. Autolt, while simple and fast, lacks visual recognition and OCR, rendering it ineffective for dynamic interfaces. Ranorex and TestComplete, although feature-rich, rely heavily on internal UI inspection, which PowerOn does not allow. Their relatively high scores in general integration and enterprise support do not translate to practical applicability here.

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

Tool	Cost	Installation	Visual Automation	AI Capabilities	Integration	OCR Accuracy	Documentation	Custom Scripting	Testing Flexibility	Security/ 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

Table 1. Tool Criteria Scores

2.4.7 Conclusion of Tool Evaluation

The evaluation of multiple automation tools highlighted a clear distinction between what is technically ideal and what is practically achievable within the constraints of the project. Among all the platforms considered, Eggplant by Keysight stands out as the most powerful and comprehensive tool for visual test automation in closed and GUI-only environments. Its AI-enhanced test model, robust image matching, advanced OCR capabilities, and enterprise-grade reporting infrastructure make it particularly well suited for large-scale, critical systems like PowerOn. Additionally, its proven track record in sectors such as healthcare and finance demonstrate its maturity and adaptability to highly regulated and complex environments.

However, Eggplant is a commercial platform that requires a substantial investment in terms of licensing, training, and infrastructure. The use of proprietary scripting language also adds a layer of complexity when onboarding new testers or integrating with external systems. Within the context of this project such investment was not feasible.

As a result, the decision was made to proceed with SikuliX, an open-source alternative that, while not as feature-rich or enterprise-oriented as Eggplant, provides the necessary visual automation capabilities in a more accessible and lightweight form. SikuliX allows full control over the automation logic using Python-like syntax (Jython), supports OCR through Tesseract, and integrates easily with other tools as CSVs and Java Swing components. Although more sensitive to UI changes and lacking some of Eggplant's analytical and management features, SikuliX is highly adaptable, fast to prototype, and fully usable without elevated system permissions.

In a different scenario (such as a full-scale industry deployment with no budget constraints) Eggplant would likely have been the preferred solution, especially if long-term maintainability, scalability, and integration with existing QA infrastructure were priorities. It offers a more robust ecosystem for continuous testing, reporting, and enterprise collaboration. That said, SikuliX has proven more than sufficient for the needs of this project, successfully automating complex user workflows in PowerOn and providing a solid foundation for future extension.

Ultimately, this project has not only implemented a practical solution using SikuliX, but also established a methodology that can be later scaled or migrated to more powerful tools like Eggplant, should the organization decide to invest further in test automation infrastructure.

3 Design and Methodology

3.1 Environment Setup and Tools Used

The successful implementation of this project required careful preparation of the working environment and selection of tools suited to the constraints of both the PowerOn system and project scope. This section outlines the technical setup, the core components used during development, and how they were configured to support the testing objectives.

3.1.1 PowerOn Access and Restrictions

All tests were developed and executed within the test environment provided by Scottish Power. This environment closely mirrors the real operational version of PowerOn but with limited administrative privilege. Importantly, access to the system is restricted post-login, meaning that all automated actions had to be performed after the user had manually launched the program and authenticated with their credentials. This constraint shaped the automation framework, which assumes the interface is active and ready for input at the moment the script begins execution.

The specific version of PowerOn available during the project was v6.9.3. This version includes multiple interface types, such as Network Diagram, GeoView, Work Package Manager, Safety Documents, Incident Management..., as it is shown in Figure 1, all of which operate entirely through graphical components. These interfaces do not expose internal object models, source code, or backend APIs, making traditional test automation tools unsuitable.

3.1.2 Tools and Libraries

The following tools and libraries were selected and configured to support the automation effort:

SikuliX 2.0.5:

The main automation tool used to interact with PowerOn. SikuliX allows the automation of GUI workflows through image recognition and simulates user behaviour via mouse and keyboard actions. It was installed and configured with its native IDE, using Jython scripting for full control of logic and flow.

SikuliX was selected because it directly fits the constraint of a closed, GUI-only environment with no public APIs or inspectable object tree, enabling interaction purely via image recognition and input simulation. In comparative testing, Eggplant offered superior model-based testing and analytics but imposed prohibitive licensing costs and proprietary language (SenseTalk) that did not align with the project's scope. Autolt's lack of built-in OCR and pixel-matching robustness made it unsuitable for highly dynamic visuals. Consequently, SikuliX provided the best cost-fitness balance under the project constraints.

Tesseract OCR (via SikuliX):

Embedded within SikuliX, Tesseract was used to perform OCR (Optical Character Recognition) on selected screen regions. This was particularly useful for validating whether an element was correctly displayed or for identifying dynamic content without relying on images.

Tesseract OCR was integrated to validate dynamic, text-based states (such as labels, scale indicators, or dialogs) that are impractical to cover with static templates alone. Alternatives were deprioritised due to cost and integration overhead relative to the project scope; leveraging Tesseract within SikuliX kept the stack cohesive and lightweight. OCR complements pixel matching, improving test oracle quality where visual similarity alone could misidentify elements.

Java Runtime Environment (JRE):

Required to run SikuliX, which is based on Java. In the working environment, 1.8.0 version of the JRE was installed alongside the SikuliX IDE.

Python:

While not essential to the core logic, Python was used during early experimentation phases, particularly when exploring integration with Robot Framework or parsing CSV logs. Ultimately, Python was not used in the final implementation due to compatibility issues with SikuliX. However, its potential for structured test management, reporting automation, and external data processing makes it a strong candidate for future extensions of the project, should the automation framework be expanded or integrated with CI/CD pipelines or testing dashboards.

CSV Files:

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.

CSV input/output decouples test logic from test data, enabling batch runs reproducibility, and auditability without modifying code. Compared with embedding data in scripts or ad-hoc spreadsheets, CSVs are portable, tool-agnostic, and simplify large-scale validation and longitudinal analysis.

The batch test campaign serves as a clear proof-of-concept, demonstrating that the framework not only executes large-scale GUI based testing consistently but also produces structured, auditable evidence for every iteration, thereby validating its effectiveness in closed, mission-critical environments.

Java Swing GUI Components:

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.

The following interactive elements were designed and embedded into the automation flow:

- **Checkbox dialogs:** Users can select which specific tests or test categories they wish to run. This modular approach allows for flexible testing without running the full suite unnecessarily.
- **Input fields:** Instead of hardcoding variables, input boxes prompt the user to enter custom data at runtime (such as location names, filter values, or test labels). This enhances adaptability and support on-demand execution with variable parameters.
- **Drop-down menus:** Where applicable, options were provided via combo boxes to constraint input choices (e.g., selecting a test mode), reducing typing errors and enforcing consistency across runs.
- **Confirmation and alert popups:** visual feedback is provided to confirm that a test was initiated, completed or encountered an issue. In the case of missing data, OCR failure, or unexpected screen states, user-friendly warnings or prompts are displayed.

These UI elements significantly enhanced the overall user experience, especially for operators with limited scripting knowledge. They also made the framework suitable for batch testing in semi-supervised environments, where a tester might need to configure a large test batch and leave the automation to run with minimal supervision. By reducing the cognitive load and technical barrier to use, the integration of swing-based dialogs improved both reliability and adoption potential within a practical industrial setting.

Lightweight Swing dialog reduced operator error and lowered the barrier to use, which is preferable to command-line flags or external GUIs in a restricted desktop environment. This improved adoption and supported semi-supervised batch testing.

The overall interaction between modules and supporting tools is illustrated in Figure 2, highlighting how data flows from user or CSV input, through the control and action layers, to the output layer where results are logged and feedback is provided.

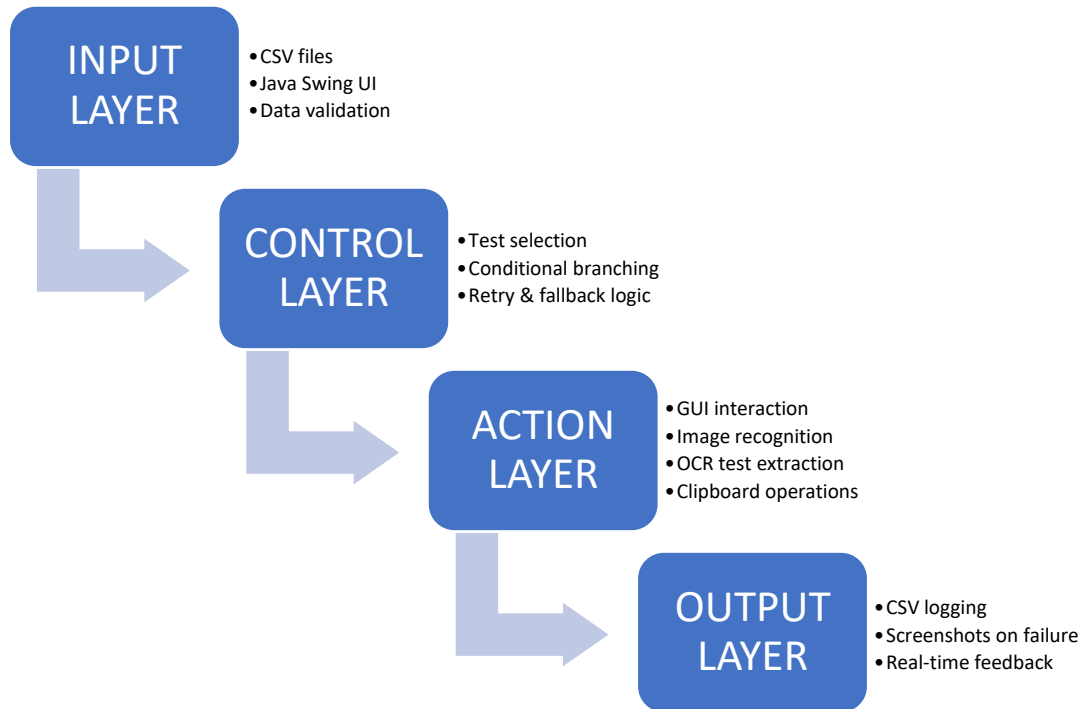


Figure 2. High-level architecture of visual automation framework

The batch test campaign exemplifies these design decisions in practice, showing that the framework can execute large-scale GUI based testing with consistent success while generating structured, auditable evidence for every iteration.

3.1.3 Development Environment

The automation scripts were developed and tested on standard company workstations running Windows, without administrative privileges. This required all tools to be portable, avoid system-level dependencies, and be easy to configure in a user-level context. All configuration files, images, and test assets were stored locally in organised folders for consistency and maintainability.

The use of image libraries, named according to interface type and element function, made the codebase easier to maintain and adapt as PowerOn evolved. This modular approach also supports potential deployment across different machines or teams in the future.

3.2 Test Automation Architecture

The automation framework developed in this project was designed with modularity, reusability, and adaptability as core principles. Rather than implementing one monolithic script, the system was built as a collection of independent, callable functions that represent distinct actions within PowerOn. This approach allows for better maintenance, easier debugging, and greater flexibility in composing new test scenarios from existing building blocks.

3.2.1 Structure and Modularity

Each function encapsulates a specific unit of behaviour, such as opening a toolbar, applying filters, searching for a location, checking if a dialog has appeared, or validating visual changes on screen. These functions are grouped into logical categories (e.g., navigation, filtering, input handling, verification) and stored in clearly named script files. This modularity supports code reuse and facilitates rapid adaption to future interface updates or new testing requirements.

To manage the complexity of automating visual interactions in a modular and maintainable way, the system is structured into four distinct layers: Input, Control, Action, Output. This architecture enables the separation of concerns and facilitates future modifications, extensions, and debugging.

Input Layer:

The input layer is responsible for capturing and validating the data required to execute tests. It operates in two primary modes:

- User-driven mode, where graphical interfaces built using Java Swing are displayed. These interfaces allow the tester to select which tests to run (via checkboxes), enter specific values such as location names (through input fields), or choose test targets from a list (using drop-down menus).
- CSV-driven mode, in which input values are loaded from external data files. This is used when running batch tests across multiple elements. For example, a list of locations, configurations, or test scenarios can be loaded dynamically without changing the code.

Before proceeding to execution, the input layer verifies that values are valid (e.g., not empty or consistent with expected format), preventing runtime errors and ensuring predictable test behaviour.

Control Layer:

This layer acts as the central brain of the framework. Based on the parameters collected from the input layer, the control logic determines:

- Which tests to run in which order, and how they should interact.
- Conditional branching, based on screen conditions, system response, or user preferences.
- Retry and recovery strategies, such as repeating a step if a required image is not found on the first attempt.
- Fallbacks, in case alternative workflows are needed (e.g., different versions of the interface or slightly changed button designs).

This layer is implemented using a combination of control structures (*if()*, *while()*) and dynamic dispatching to reusable functions. It ensures that automation can proceed even when facing variation in the interface or occasional delays.

Action Layer:

The action layer is responsible for carrying out the actual interaction with the PowerOn interface. It uses SikuliX commands to:

- Simulate human behaviour: Using *click()*, *type()*, *dragDrop()*, *hover()*, or *scroll()* to interact with buttons, fields, and panels.
- Visually detect interface elements: With *exists()*, *wait()*, or *waitVanish()* functions, comparing screen content with pre-captured reference images at controlled similarity thresholds.
- Extract or verify screen content: Using OCR (via Tesseract) to read text or numerical values from interface regions, such as name locations, scale indicators, or alert messages.
- Insert data efficiently: In some cases, *type()* command was replaced by clipboard operations (*paste()*), which not only speeds up execution but also makes the automation less human-like and more robust.

The logic in this layer is designed to be atomic, so each function performs one discrete operation that can be logged, repeated, or skipped as needed.

Output layer

The final layer is dedicated to recording the outcome of each test and providing useful feedback to the user. It supports two main mechanisms:

- Structured result logging: After each test step or batch cycle, outcomes are written to a CSV file. Each row includes metadata such as the timestamp, test type, input value, result (e.g., "Success", "Failed-Image not found"), and optionally a screenshot of the failure state. This format supports traceability, result aggregation, and further analysis.
- Real-time feedback: Popups, alert dialogs, or status messages are used during testing to indicate whether a step has completed, failed, or requires user intervention. These messages help ensure that semi-supervised test runs can be monitored without the need from a live debugger.

In future versions, this layer could be extended to generate graphical reports or connect to external dashboards via APIs, should the system be integrated with a larger testing framework or CI/CD toolchain.

3.2.2 Image-Based Logic and Handling

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

Because 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.

Even minor differences, such as anti-aliasing effects on text, slight blurring, or compression artefacts, can lead to a failed match. In some instances, some pixels shift or minor colour deviation was enough to break recognition, especially when similarity thresholds were set high to avoid false positives. To address this, images were captured:

- With pixel-level precision, using tools like SikuliX’s built-in screenshot selector, highlighted in Figure 3.
- From a maximised and isolated interface, ensuring that no overlapping windows or tooltips distorted the reference region.
- In idle states (e.g., when buttons were not hovered or active), to ensure baseline visuals.

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.

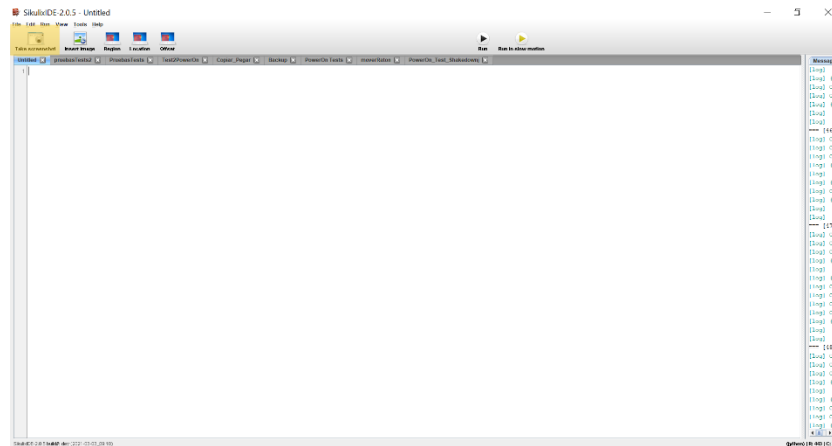


Figure 3. SikuliX screen. Take screenshot tool highlighted

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 offered several benefits:

- **Maintainability:** As PowerOn evolves or changes UI components (after updates), developers can easily identify which tests use which images and update only the affected files.
- **Scalability:** As the number of automated test cases grows, so does the number of required image assets. Structured storage prevents disorganisation and duplication, making it easier to navigate the library and apply version control.
- **Reusability:** Shared assets, such as popups, toolbar buttons, or loading icons, can be reused across different test scripts, reducing redundancy.
- **Debugging and refactoring:** When a test fails due to an image mismatch, its location in the directory gives immediate context on its origin and purpose, speeding up debugging and image replacement.

This system forms the visual backbone of the entire automation framework. In future implementations, it could be enhanced with metadata tagging or a simple asset management UI to further streamline updates and organisation.

Fallback logic

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

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. For example, a toolbar button may appear in one form during a normal session, and in a slightly different form when running under admin mode or using an alternative colour scheme.

- 4 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. This is shown in

This Appendix C Alignment with United Nations Sustainable Development Goals (SDGs), particularly in the context of sustainable energy systems, innovation in industry, and operational resilience in critical infrastructure.

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.

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.

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.

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.

SDG 13: Climate Action

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 BEFO

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.

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

Appendix B: Single Tests Code and Appendix C: Batch Tests Code.

This strategy dramatically improved test reliability across sessions, machines, and environments. While it introduced more image assets to manage, the gains in compatibility and execution success outweighed the cost in maintainability. Future improvements could include the automatic detection of environment profiles to preselect likely variants and streamline detection.

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.

- For precise and static interface elements (such as fixed-position toolbar icons or buttons that always appear in the same place and form), high similarity thresholds (typically between 0.80 and 0.99) were used. These elements are not expected to vary between sessions, and therefore require a close visual match to avoid detecting incorrect elements.
- For semi-static elements with minor variations in shading, resolution, or background medium similarity thresholds (0.60 to 0.79) are used. It balances tolerance for small visual changes while keeping detection reliable.
- In contrast, for repeated, floating, or highly variable elements (such as markers in a diagram, text fields with different lengths, or background-specific shapes) a lower similarity threshold (around 0.30 to 0.50) was often more appropriate. These elements might appear in different parts of the screen or with subtle differences in colour, font rendering, or zoom level. Lower thresholds helped detect general patterns even when the visual was partially obscured or distorted.

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 1, 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.

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 BEFO

Where possible, the use of surrounding context, such as additional reference points or screen structure, was used to increase confidence in a detection. This reflects the reality of testing closed systems like PowerOn, where visual automation must accommodate variation while still delivering reliable results.

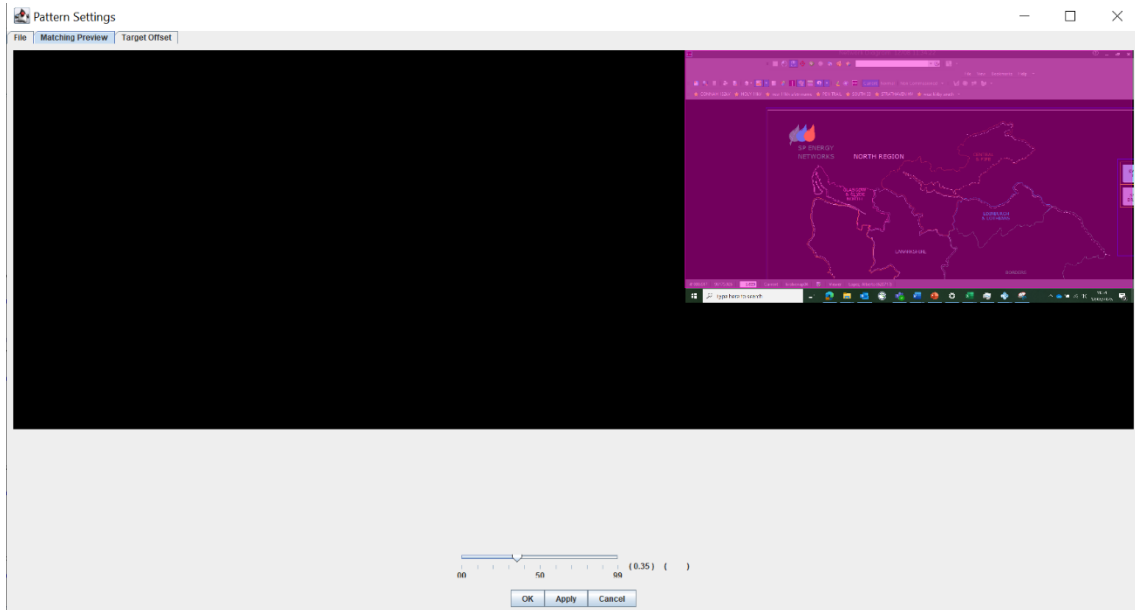


Figure 4. Matching settings example

Sequential State Validation

A key challenge in GUI automation is ensuring that the system has reached the correct state before proceeding. Relying only on the appearance of one expected image is often insufficient, especially in asynchronous systems where UI elements may load progressively, overlap temporarily, or be obstructed by popups.

To counter this, 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

Since 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

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

from hanging indefinitely in cases where an expected element fails to appear due to error, delay, or user interference.

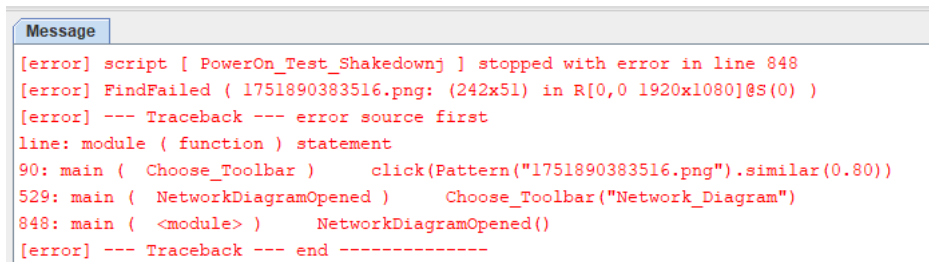
Timeout durations are chosen based on the expected load time of each interface component. For example:

- Simple images: 0.5 – 2 seconds.
- Diagram loads or interface switches: 30 – 120 seconds.
- OCR verification or dynamic list refresh: 10 seconds.

If the timeout expires and the image is still not found, the framework logs the failure and either:

- Continues to the next test item, if the step is optional, or it is batch mode, where it will give the error at the results.
- Halts the execution and displays a popup message, if the step is critical, similar to the Figure 5 example.

This design allows the framework to maintain momentum during batch testing while still collecting meaningful diagnostics for each case. Timeout values are stored as global variables, making them easy to adjust if performance conditions change.



```

Message
[error] script [ PowerOn_Test_Shakedownj ] stopped with error in line 848
[error] FindFailed ( 1751890383516.png: (242x51) in R[0,0 1920x1080]@S(0) )
[error] --- Traceback --- error source first
line: module ( function ) statement
90: main ( Choose_Toolbar ) click(Pattern("1751890383516.png").similar(0.80))
529: main ( NetworkDiagramOpened ) Choose_Toolbar("Network_Diagram")
848: main ( <module> ) NetworkDiagramOpened()
[error] --- Traceback --- end -----
  
```

Figure 5. Image not found error message

Scalability Considerations

One of the goals of this project was to ensure that the automation scripts could be executed not just on one machine, but across multiple workstations or users within Scottish Power. This introduced the need for platform-agnostic design wherever possible.

To that end, the following guidelines were adopted:

- The automation scripts use maximised window mode in PowerOn to ensure consistent screen coordinates and layout.
- No interface scaling should be applied. The system is calibrated for 100% scaling.

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

- Fonts, themes, and OS visual settings should remain at defaults where possible.
- SikuliX settings, including similarity thresholds and timeouts, can be tuned per machine if variability is detected.

Even with these precautions, image-based automation will always retain some environmental sensitivity. As such, this framework includes tools and practices for quick image replacement, fallback matching, and modular testing, enabling adaptation with minimal effort. These practices make it not only usable in the current environment but sustainable and expandable in future deployments.

4.1.1 Scalability for Batch Testing

One of the 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. [16] [17]

This means the system is capable of growing organically with minimal technical overhead, adapting to the evolving needs of business. As the RTS team adds new network regions, tools, or operational scenarios, these can be incorporated into automated validation routines in minutes. The modularity also ensures that test cases do not become unwieldy or redundant at dataset scales, maintaining clarity and performance even with large batches.

Reusability

Because 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.

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

For example, a search function developed for Network Diagram mode can also be reused for GeoView mode with only a change in input parameters. Similarly, the same logic for applying filters can be reused across various test targets, allowing for flexible and dynamic test assembly. Over time, this leads to a library of reusable, well-tested components that can be recombined to create new test scenarios with minimal effort, supporting both prototyping and production testing.

Parallelisation Potential

Although 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 or identifier.
- Test category.
- Result status (Pass/Fail/Warning).
- Error type or exception message (if applicable).
- 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 6 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.

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 BEFO

Substation	Result	Error	Failed Step	Screenshot	Duration	Timestamp
AYR GSP	Pass				15.432	10/08/2025 11:18
BONNYBRIDGE GSP	Pass				9.049	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				9.416	10/08/2025 11:19
COATBRIDGE A GSP	Pass				9.833	10/08/2025 11:19
COYLTON GSP NEW	Pass				9.915	10/08/2025 11:19
CROOKSTON 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				10.321	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: 175199205482	Open scale	C:\Users\B620717\Desktop\PowerOnResults\snap_auto_fail_20250810_112116.png	13.1	10/08/2025 11:21
GLENCLUE GSP	Fail	[Open filters] Cannot click; not found: 175206469571	Open filters	C:\Users\B620717\Desktop\PowerOnResults\snap_auto_fail_20250810_112123.png	6.595	10/08/2025 11:21
GLENCLUE GSP	Pass				15.15	10/08/2025 11:21
GRANGEMOUTH A GSP	Pass				9.099	10/08/2025 11:21
GRANGEMOUTH C GSP	Pass				9.024	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 G	Pass				9.176	10/08/2025 11:22
KILBOWIE GSP	Pass				8.667	10/08/2025 11:22
KILLERMONT GSP NEW	Pass				9.25	10/08/2025 11:22
KILMARNOCK SOUTH G	Pass				9.355	10/08/2025 11:23
KILWINNING GSP	Pass				9.259	10/08/2025 11:23
LINN MILL GSP	Pass				9.212	10/08/2025 11:23
LIVINGSTON EAST GSP	Pass				9.287	10/08/2025 11:23
MAYBOLE GSP	Pass				8.395	10/08/2025 11:23
NEARTHILL GSP (NEW	Pass				9.214	10/08/2025 11:23
NEWTON STEWART GSP	Pass				9.084	10/08/2025 11:24
PARTICK GSP	Pass				9.55	10/08/2025 11:24
RAVENS CRAIG GSP	Pass				9.097	10/08/2025 11:24
SALT COATS B	Pass				9.21	10/08/2025 11:24
ST ANDREW'S CROSS G	Pass				8.464	10/08/2025 11:24
STIRLING GSP	Pass				8.981	10/08/2025 11:24
STRATHAVEN GSP	Pass				9.162	10/08/2025 11:24
STRATHKELVIN GSP	Pass				9.175	10/08/2025 11:25
TOMGLAND GSP (T)	Pass				9.917	10/08/2025 11:25
WEST GEORGE ST GSP	Pass				9.962	10/08/2025 11:25
WESTBURN ROAD GSP	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
WISHAW GSP	Pass				9.43	10/08/2025 11:26
WISHAW GSP A	Pass				8.579	10/08/2025 11:26
WISHAW GSP B	Pass				9.458	10/08/2025 11:26

Figure 6. Results CSV example

This batch capability was particularly valuable in the context of network-wide testing, where it was essential to verify whether PowerOn's behaviour remained consistent across different regions, asset types, or interface contexts. For example, a filter that works correctly on one substation might behave differently in another due to data discrepancies or configuration issues. With CSV-driven input, such anomalies can be detected systematically and at scale.

In short, this batch execution model transforms the framework from a tool for isolated test cases into a scalable system for dataset-wide validation.

4.1.2 Error Handling and Watchdog Logic

In GUI-based automation systems, especially those dependent on visual recognition rather than programmatic interfaces, unexpected behaviours and inconsistencies are not only possible but common. PowerOn, as a complex and asynchronous SCADA platform, introduces multiple sources of variability: variable loading times, inconsistent window positions, warning popups, and visual differences due to environmental factors (such as screen resolution or theme). To ensure robust execution in such conditions, this framework integrates a comprehensive error-handling strategy.

The error handling architecture is built upon a combination of structured exception management, preventive conditional checks, and a dedicated watchdog mechanism to handle non-critical disruptions.

Structured Exception Handling

ICAI ICADE CIHS
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 BEFO

Each major automation block is wrapped in structured “try/except” constructs. These catch common execution failures, such as the absence of an expected image, a failed click, or invalid OCR output, and prevent the script from crashing entirely. Instead of terminating the whole test session, the system:

- Logs the failure condition in the output file, including the input value and type of error.
- Takes an optional screenshot of the screen at the point of failure for debugging.
- Moves on the next input or scenario, depending on the severity of the error and the test configuration.

This ensures that a single failure does not compromise an entire batch run, and that test coverage remains as complete as possible even when individual steps encounter problems.

Conditional Image Checks

Rather than assuming interface elements will always be present, the framework uses *exists()* or *wait()* functions to verify conditions before acting. This includes checking:

- Whether a button or menu is visible before attempting to click.
- Whether a dialog has fully loaded before interacting with it.
- Whether a previous window has disappeared before continuing to the next step.

These conditional checks are particularly useful in dynamic interfaces where the timing of UI rendering varies based on the dataset and the system load. Incorporating timeouts and fallback behaviours ensures the framework can handle such variability gracefully.

Watchdog Functionality

Unexpected windows, such as confirmation dialogs, warning messages, or license popups, can easily interrupt a test flow if not handled promptly. To address this, the system includes a watchdog mechanism, which runs at predefined checkpoints and scans the screen for known disruptive elements [18] [19]. When detected, the watchdog:

- Identifies the popup using image-based detection.
- Executes the correct dismissal action (e.g., clicking “OK”, “Cancel”, or closing the window).
- Waits for the screen to return to its previous expected state before resuming the test.

This proactive monitoring allows the automation to remain resilient in the face of non-critical but potentially blocking UI events. The watchdog can be expanded with new visual patterns as more edge cases are discovered.

Delayed Response Handling

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 BEFO

PowerOn frequently loads data-heavy elements which can take from seconds to minutes to appear. To avoid acting too early, the framework uses synchronisation strategies, including:

- *wait()* to pause until a required image appears.
- *waitVanish()* to confirm the disappearance of loading spinners or transitions.
- *sleep()* timeouts where interface behaviour is inconsistent.

This dynamic timing model reduces reliance on hardcoded delays and increases adaptability across sessions and machines.

Overall, the combination of structured error control, dynamic screen checks, and watchdog routines results in a fault-tolerant, self-recovering automation system. This approach minimises brittle failure points, supports unattended execution of long test cycles, and increases trust in test outcomes, making it suitable for real-world industrial use.

5 Case Study

5.1 Workflow Example: End-to-End Test Case

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.

5.1.1 Step-by-Step Description

1. Launch and Log in (Manual Precondition)

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. Collecting the 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 7.
- 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.

The input is checked for validity and formatted before being passed to the search logic.

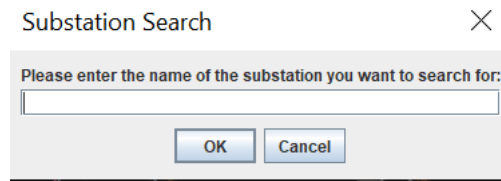


Figure 7. Popup dialog for manual testing

3. Verifying Toolbar Access

The PowerOn toolbar is essential for accessing different modules (Figure 8). 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 (to account for visual variations).
- If found, it clicks the symbol to open the toolbar.
- If the toolbar is pinned (fixed to the screen), the script clicks the icon to unpin it, ensuring it does not obstruct other elements during the test.

This step guarantees a predictable UI layout and prevents accidental interference with navigation or pattern matching later in the workflow.

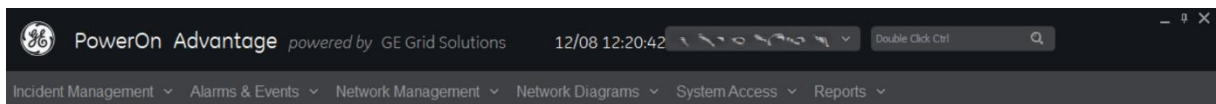


Figure 8. PowerOn toolbar

4. Opening the Network Diagram Interface

With the toolbar 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:

- The interface has fully loaded (as in Figure 9).
- No overlay windows (loading messages or popups) remain visible.
- The view is maximised (if not, the script maximises it).

In this step, the script uses *wait()* and *waitVanish()* to synchronise with PowerOn's dynamic loading times, ensuring no test steps are executed prematurely.

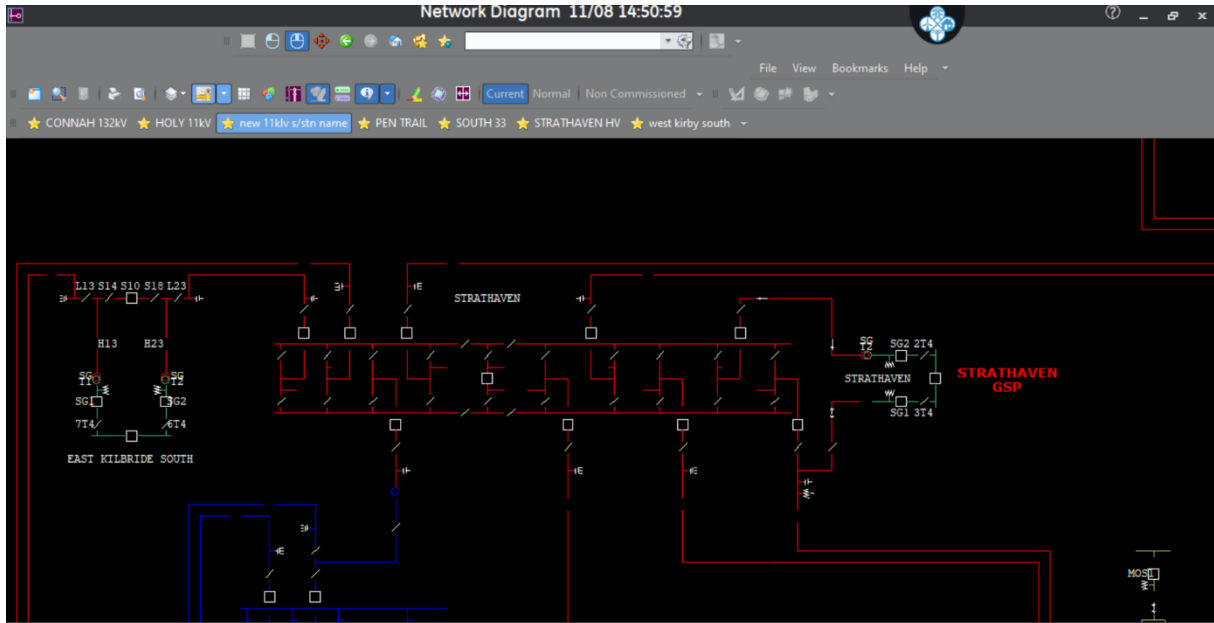


Figure 9. Network Diagram

5. Preparing the Diagram Environment

Before performing the search, the system applies a predefined set of diagram filters, with the settings shown in Figure 10; to ensure only relevant assets are shown:

- Voltage levels (Low Voltage, High Voltage) are toggled depending on the test type.
- Location types (Transmission, Sub Transmission, Grid, Primary, Secondary) are enabled or disabled.
- Visibility filters and search modes are adjusted to broaden or narrow the result scope.

Each filter action is validated using pattern matching, with image similarity thresholds tuned to recognise both active and inactive states of the UI icons. This guarantees that the test operates on a consistent and controlled visual environment.

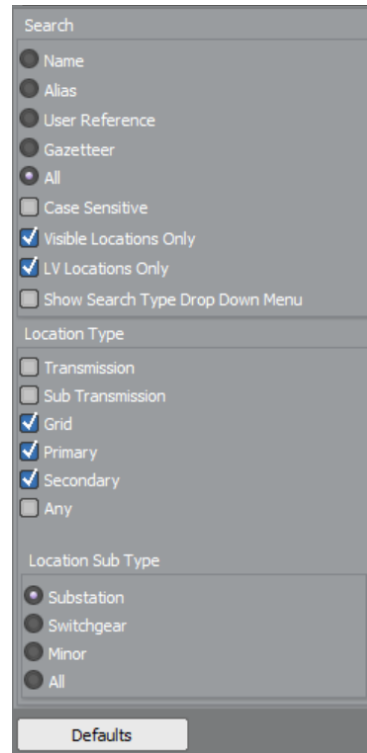


Figure 10. Filter settings

6. Searching 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. Verifying 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.

The system then verifies that key graphical markers appear on screen, such as:

- Location symbols.
- Labels.
- Selection highlights.

- Visual outlines.

This is done using precise image matching in known positions or regions of interest, confirming that the asset has been correctly identified and loaded, as it was done in the example from Figure 4.

8. Switching to GeoView and Repeating Validation

With the Network Diagram test complete, 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 11), requiring separate reference images and detection logic. The framework handles this by using mode-specific image sets and conditionals to execute the correct steps.

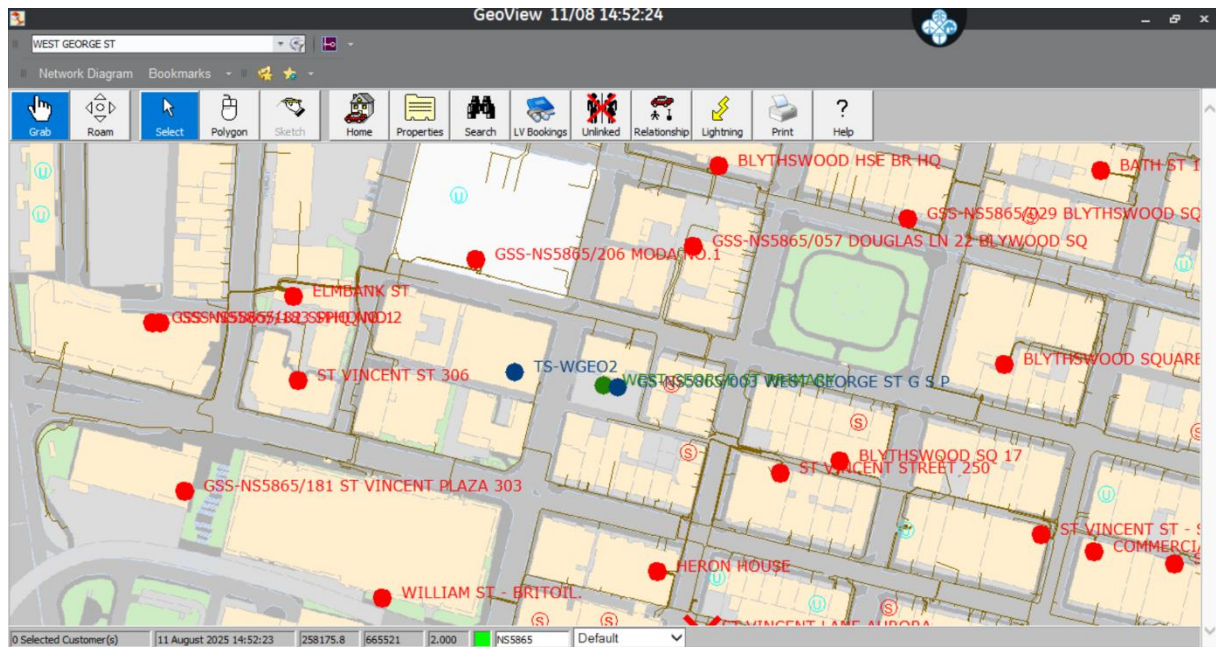


Figure 11. GeoView Diagram

9. Logging 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 the following fields:

- Test inputs (e.g., location name).
- Test mode (e.g., Network Diagram, GeoView).
- Result (e.g., Pass, Fail).
- Optional error message or failure reason.
- Timestamp of execution.

- 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 the test is running 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.

5.2 Results Logging and Reporting

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.

5.2.1 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, and includes the following fields:

- **Tested item:** Identifies the specific input under test (typically the name or unique identifier of a substation, job entry, geographic location, or asset). This field is essential for correlating results to actual operational entities in PowerOn and for tracking which inputs produce consistent or inconsistent behaviour across test runs.
- **Test category:** A label describing the nature of the text executed. Examples include "Network Diagram", "GeoView", "Work Package Manager", "Trace", "Safety Documents", "OMS", "PC Call Supervisor", "CallTaker", "Trends", or "Alarms".

Categorisation allows test engineers to group, sort and filter results by type, facilitating issue tracking and regression coverage analysis.

- **Test subcategory:** A label describing which subcategory of the main categories the test has developed. Some of these subcategories are “Substation Search”, “Incident Search”, “Display Customers”, “Fault Search”, “Job Search”, “Schedule Search”, “Permit for Work Search”, “Sanction for Test Search”, “General Consent Search”, “SOP Search”, “Operation Restriction Search”, or “Technical Limitation Record Search”. This categorisation allows to group, sort and filter results by smaller classes, facilitating issue tracking and regression coverage analysis.
- **Result status:** High-level classification of test outcome. Typical values are “Pass” (all expected behaviours were observed, and no anomalies occurred), “Fail” (one or more critical steps could not be completed as intended), or “Partial” (some components succeeded, but the test could not be fully validated). This field is designed to be machine-readable, enabling automated statistical aggression (e.g., calculating passes rates or test stability over time).
- **Timestamp:** Exact time and date at which the test was completed. This allows results to be chronologically ordered, compared across different sessions, and correlated with known events (PowerOn updates, system slowdowns, or user interventions).
- **Error description:** If a test fails or returns a partial result, this fields contains a short message indicating the root cause or symptom. Some examples are: “Toolbar icon not found”, Search field did not appear”, “OCR mismatch on location label”, or “Loding timeout exceeded”. These error strings help testers quickly identify and troubleshoot failures without having to re-run the script or manually inspect screenshots.
- **Screenshot path:** The system captures a screenshot of current screen at the point of failure and stores it in a predefined directory. The path to this file is logged in this field. This feature is particularly useful for debugging visual mismatches, validating system state during regressions, and building a visual history of interface behaviours.

By logging this structured information, the framework produces self-contained, transparent, and traceable test artifacts. These logs can be:

- Reviewed manually by test engineers to investigate individual failures.
- Parsed automatically by scripts to generate metrics or dashboards.
- Exported and versioned for audit or regulatory purposes.
- Used to compare test results over time, identify flaky behaviours, and prioritise fixes.

The simplicity and modularity of the CSV format also ensure that additional fields (such as execution duration, system CPU usage, user identifier, software version) can be added later with minimal impact on the existing system.

In summary, the logging structure transforms the automation framework from a runtime tool into a long-term quality assurance asset, supporting data-driven decisions and systematic process improvement within Scottish Power's RTS operations.

5.2.2 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:

1. On-Screen Popup Messages (Java Swing Dialog)

Throughout execution, the system generates interactive popup messages using Java Swing components. These popups are triggered at key decision points or status updates, such as:

- Successful completion of a test step.
- Entry errors (such as "No location selected").
- Warnings when a visual element is not found within the expected timeframe.
- Notifications that fallback or retry path is being used.
- Confirmation that a batch test is starting or has been completed.

These messages are intentionally kept brief and clear, using plain language to help non-technical users understand the system's status without needing to inspect logs or code, such as Figure 12. In supervised runs, this gives the tester the opportunity to intervene or cancel if a critical issue is detected early.

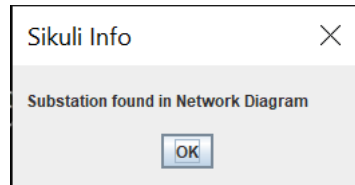


Figure 12. On screen popup message

2. Printed Console Logs

In addition to graphical notifications, the system outputs real-time execution traces to the console or terminal window. These logs include:

- Function calls and parameters.
- Timing information (such as start/end of major steps).
- Similarity thresholds used in image detection.
- OCR results and confidence.
- Internal flags and fallback conditions triggered.
- Test outcome summaries.

These printed logs are particularly valuable for developers and testers during debugging, as they provide visibility into the internal decision-making process of the automation engine, like Figure 5. They also help reproduce errors, tune performance, and analyse execution timing without relying solely on post-mortem logs.

3. Execution Counters and Batch Progress Indicators

For batch test sessions, the framework maintains execution counters which track:

- The total number of entries in the input CSV.
- The current position within the batch.
- The cumulative number of successes and failures.
- The estimated time to completion.

This data is surfaced either via printed logs or periodic popup updates. In long test runs, such as when validating hundreds of locations or performing regression sweeps, this allows the user to confidently monitor the automation without constant supervision.

4. 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. This alert includes:

- The error message or failure description.
- A prompt indicating whether the test should stop, skip to the next item, or retry.

These alerts are essential in semi-automated environments, where full autonomy is not yet in place. They help prevent silent failures or inaccurate results and give the user control over how to proceed.

5. Benefits of Real-Time Feedback

This layered feedback model offers significant operational advantages:

- **Transparency:** Users are not left guessing what the script is doing or whether it's stuck.
- **Responsiveness:** Failures can be reacted to immediately, reducing wasted time or inaccurate results.
- **Trust and usability:** Non-developers can use the tool with confidence, thanks to human-readable feedback.
- **Debugging efficiency:** Developers and testers can locate failure points more easily, accelerating fixes.
- **Scalability:** In batch or future CI/CD applications, these mechanisms support unattended operation with minimal risk.

5.2.3 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.

The result data can be filtered and aggregated to answer key operational and research questions such as:

- **What is the overall success rate of the automation system?** This helps determine the reliability of both the test logic and the underlying PowerOn functions being exercised.
- **Which locations or inputs consistently fail?** By grouping results by input, it is possible to detect whether certain parts of the network or dataset are problematic, which may point to underlying configuration or data integrity issues.

- **Are failures linked to specific modules?** Aggregating by test category can reveal patterns in system behaviour and identify modules that are more fragile or inconsistent across sessions.
- **How does test performance evolve over time?** Timestamp-based analysis can highlight regressions introduced by PowerOn updates, improvements after bug fixes, or degradation due to environmental changes.

These aggregated insights support data-driven quality assurance, allowing RTS engineers to prioritise fixes, evaluate the impact of software updates and continuously improve test coverage.

To support this long-term objective, future versions of the framework could incorporate:

- Automated generation of summary reports.
- Dashboards or visual analytic tools that present real-time or historical trends visually, allowing managers and testers to interpret complex datasets quickly and accurately.
- Continuous integration (CI) compatibility, where test runs are triggered automatically as part of software delivery pipelines, and results are fed into monitoring systems for live validation.

Through this structure, the framework not only executes tests but also becomes a platform for system observability, reporting, and strategic improvement.

5.2.4 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. This makes it easy to:

- Locate and retrieve past test results.
- Compare outputs across test cycles.
- Identify historical trends in failures 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.

The logging mechanism is also fully extensible by design. New fields such as execution duration, CPU usage at test time, test execution environment, or confidence scores from OCR or image matching can be added to the output with minimal impact to existing scripts. This makes the system future-proof, enabling integration into more advanced QA processes, benchmarking studies, or academic research scenarios.

5.3 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.

5.3.1 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 underlying data structures, all interactions are performed through screen-based logic using SikuliX. This introduces several sensitivities:

- Screen resolution and scaling must remain constant between sessions, as even minor visual shifts can cause image mismatches.
- The framework assumes a 100% display zoom and a standardised interface layout (maximised windows).
- Any change to the UI, such as updated icons, altered themes, or unexpected popups, can invalidate reference images and cause failures.

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.

5.3.2 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 includes:

- Retaking affected screenshots.
- Replacing them in the correct folder structures.
- Validating that the new images do not introduce false positives.

This creates a maintenance cost proportional to the number of test cases and images used. Over time, unless managed properly, this can affect scalability.

Some future solutions might involve:

- Automating image verification.

- Building tooling to flag broken image matches.
- Migrating toward hybrid systems that combine visual and data-based logic where possible.

5.3.3 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.

As a result, the automation framework:

- Cannot access underlying databases, meaning it cannot validate whether a user action correctly updates a backend record unless such a change is visibly reflected in the UI.
- Cannot track system logs, memory states, or execution traces, which are often required to detect subtle bugs, race conditions, or improper exception handling.
- Cannot detect system errors or inconsistencies unless they manifest through visual cues, such as a warning message or missing label.

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.

In practice, this means the automation should be used in conjunction with complementary backend validation tools. This layered approach offers interface confidence and data integrity assurance.

5.3.4 Performance Variability and System Load

The performance of PowerOn is affected by a range of runtime 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.

To mitigate this, the framework uses `wait()`, `waitVanish()`, and flexible timeouts to synchronise actions with the visual state of the screen. However, despite these precautions, flaky test behaviour can still occur in edge cases.

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.

For more robust performance in production or CI/CD environments, the automation could be paired with resource monitoring tools or extended to dynamically adjust wait times based on system load feedback.

5.3.5 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:

- The user must launch PowerOn manually and enter their credentials, as the login interface is protected against automation and includes unpredictable graphical security elements.
- Once the system is running, some UI elements require manual dismissal if they are not recognised by the watchdog logic.

These preconditions limit the framework’s ability to operate in a fully autonomous fashion, particularly in unsupervised overnight or server-based runs. Additionally, they prevent direct integration into CI/CD pipelines or test-on-deploy architectures.

In future iterations, integration with enterprise access tools or improvements to the watchdog system could further reduce manual steps and allow full end-to-end automation.

5.3.6 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.

There is potential to extend the system with:

- Basic AI features such as fuzzy template matching or adaptative thresholds.
- Model-based UI detection.
- Test scenario generation using LLMs or reinforcement learning.

Such additions, however, require a significant architectural evolution.

In summary, despite these limitations, the system achieves its core goal: demonstrating that modular, visual test automation is possible, scalable, and valuable in closed SCADA systems like PowerOn. The outlined constraints do not undermine the approach, they clarify the boundaries within which the tool can be reliably used, and the point the way to future technical evolution.

6 Results 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 key metrics such as:

- **Results:** Whether the test passed or failed.
- **Error message:** Detailed explanation of the failure, if any.
- **Failed step:** The specific step in the test procedure that did not complete as expected.
- **Screenshot:** A captured image at the moment of failure, allowing for visual confirmation of the issue.
- **Execution duration:** The total time taken to run the test for a given substation.
- **Timestamp:** The exact date and time the iteration was executed.

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.

6.1 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 found" (6/12). Every iteration involved:

1. Launching and configuring the Network Diagram interface.
2. Applying the appropriate filters.
3. Searching for the specific substation.
4. Confirming its presence and accessibility in the interface.

5. Recording execution time, step-by-step status, and any encountered errors.

The results obtained were as follows:

- **Total test executed:** 200.
- **Successful executions (Pass):** 188 (94%).
- **Failed executions (Fail):** 12 (6%).
- **Average execution time (all cases):** 9.79 seconds.
- **Average execution time (successful only):** 10.15 seconds.

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

6.2 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.

6.3 Failure Analysis

All 12 failures fell into two image-recognition categories, as show in Table 2. 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.

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.

Table 2. Error messages from batch test

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.

6.4 Observations on Robustness and Error Handling

A key strength observed in the 200-run campaign is the framework's error-recovery and evidence-capture design:

- Batch continuity: Failures (12/200) did not terminate the run; the watchdog skipped the failing entry and proceeded to the next substation.
- High pass rate and stable timings: 94% success (188/200); median runtime 9.79 seconds.
- Minimal overhead on failure path: Failed runs tended to fail fast rather than add delay (median 9.79s vs 10.15s for successful runs).
- Bounded failure modes: All failures fell into two image-recognition steps ("Click scale – not found" (6/12) and "Open filters – not found" (6/12)).
- Structured, auditable evidence: Each failed run recorded the failed step, error message, screenshot, timestamp, and the standard CSV fields (result and duration), enabling precise troubleshooting without rerunning the batch.

Together, these behaviours, high pass rate, stable timings, graceful degradation on error, and auditable logs, provide concrete evidence that the automation framework can reliably interact with PowerOn's GUI at scale for the Network Diagram workflow.

6.5 Implications for Future Testing

From an operational standpoint, the results suggest that the developed automation framework is already suitable for large-scale repetitive testing in environments similar to the one evaluated. With minimal adjustments, particularly in the image matching similarity thresholds or wait-time tuning, the failure rate could likely be reduced to near-zero.

Moreover, given the reproducibility and speed of the batch execution process, this method can be integrated into regression testing cycles whenever updates are deployed to the target system, ensuring continuous quality validation without excessive operator time investment.

7 Opportunities for Future Work

Building on the findings discussed in Results Analysis, 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 following sections outline several key areas where this framework can evolve in future iterations, whether as part of continued development within Scottish Power or academic follow-up projects.

7.1 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.

7.2 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. These include:

- Job creation and editing workflows, which involve multiple interface components, data input fields, and visual confirmations.

- Warnings and notification handling, where the system must correctly detect, display, and log system-level alerts. Verifying the appearance, persistence, and resolution of such notifications would enhance system safety and operator confidence.
- Switching operations or isolation planning, critical from a network control perspective, often rely on precise interactions across multiple views.
- Advanced visual layers, such as overlays, contour maps, or fault trace views, which may be sensitive to performance and visual alignment.
- Performance over time, through long-running automated sessions that simulate real operator workloads and detect UI degradation or instability under load.

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.

7.3 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.

Opportunities for integration include:

- Accessing PowerOn logs or event queues to confirm that GUI actions produce expected backend response.
- Cross-verifying outputs, such as checking whether an asset marked as “edited” on screen also shows a timestamp.
- API-based probing or database lookup.

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.

7.4 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.

7.5 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.

For example:

- After a software patch or system update, a batch test suite could be triggered automatically to verify that critical functions remain operational.
- Nightly test runs could be executed on staging environments to detect regressions or inconsistencies before the system is promoted to production.
- Test results could be tied to build/version numbers, allowing developers to trace when and where specific issues were introduced.

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.

8 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 the PowerOn SCADA environment.
- A validated methodology for batch execution with robust error handling and detailed result tracking.
- Insight into current limitations (such as image similarity sensitivity) 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.

9 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] L. Ardito, G. Procaccianti, M. Torchiano and M. Morisio, "An empirical investigation on the feasibility of using GUI testing techniques for business-critical application," Information and Software Technology, 2015.
- [5] A. Rauf and S. Ali, "GUI Test Automation: A Systematic Mapping and Review," 2018.
- [6] Keysight Technologies, "Eggplant FUnctional User GUide," Keysight, 2023. [Online]. Available: <https://docs.eggplantsoftware.com/>.
- [7] SikuliX Developers, "SikuliX Documentation," SikuliX, 2023. [Online]. Available: <https://sikulix.github.io/>.
- [8] T. Yeh, T.-H. Chang and R. C. Miller, "Sikuli: using GUI screenshots for search and automation," 22nd Annual ACM Symposium on User Interface Software and Technology, 2009.
- [9] R. Smith, "An Overview of the Tesseract OCR Engine," Ninth International Conference on Document analysis and Recognition, 2007.
- [10] V. Garousi and W. Afzal, "Visual GUI testing in practice: An extended industrial case study," 2020.

- [11] S. A. Jolly, V. Garusi and M. M. Eskandar, "Automated unit testing of a SCADA control software: An industrial study based on action research," IEEE 5th International Conference on Software Testing, Verification and Validation, 2012.
- [12] Keysight, "An Enterprise Approach to Modernizing Test Automation Strategy".
- [13] Autolt Consulting Ltd., "Autolt v3 Documentation," Autolt, 2022. [Online]. Available: <https://www.autoitscript.com/autoit3/docs/>.
- [14] Ranorex GmbH., "Ranorex Studio User Guide," Ranorex, [Online]. Available: <https://support.ranorex.com/hc/en-us>.
- [15] SmartBear Software, "TestComplete Documentation," SMARTBEAR, 2023. [Online]. Available: <https://support.smartbear.com/testcomplete/docs/>.
- [16] IEEE Standards Association, "IEEE Standard for Software and System Test Documentation," IEEE Std, 2013.
- [17] G. J. Myers, C. Sandler and T. Badgett, "The Art of Software Testing," Wiley, 2011.
- [18] W. T. Wang and X. Bai, "A GUI regression testing method based on dynamic event flow analysis," Third International Conference on Software Testing, Verification and Validation Workshops, 2010.
- [19] B. P. Lamanha and M. P. Usaola, "Automated testing in a GUI based application," Software Quality Journal, 2010.

10 Appendix C: 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.

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.

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.

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.

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.

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.

11 Appendix B: Single Tests Code

```
import csv
import os
import time

from javax.swing import JFrame, JCheckBox, JButton
from java.awt import GridLayout

# Next are necessary for copy/paste
from java.awt.datatransfer import StringSelection
from java.awt import Toolkit

# VARIABLES
Wait_time = 0.5 # Wait time to find an image
Wait_time2 = 2 # Wait time when it is necessary to wait for the program to load
Wait_time3 = 0.01 # Fast actions
Wait_time4 = 1
Wait_time5 = 10
Wait_time6 = 120 # For searching options in documents
Wait_time7 = 30 # For searching

symbol_moved = 0 # PowerOn symbol has not been moved yet
GeoViewScaleChanged = 0 # 0 if not changed, 1 if changed
WPMMovedCheck = 0 # Check if Work Package Manager moved

substation_list = "C:\Users\B620717\Desktop\ejemplo.csv" # Path to substation datasheet

# FUNCTIONS

# Copy and paste the required text (instead of typing)
def copy_to_clipboard(Text):
    clipboard = Toolkit.getDefaultToolkit().getSystemClipboard()
    selection = StringSelection(Text)
    clipboard.setContents(selection, None)

def run_for_duration(duration_secs, action, stop_if_found=None):
    start_time = time.time()
    while time.time() - start_time < duration_secs:
        if stop_if_found and exists(stop_if_found, 0):
            break
        action()
        wait(0.2)

def PressDownWhile():
    type(Key.DOWN)
    # while not exists(Pattern("1752583265800.png").similar(0.95), 0):
    # type(Key.DOWN)
    # wait(Wait_time)
```

```
# Move PowerOn symbol
def Move_Symbol(): # Change time
    global symbol_moved

    if exists("1751877757141.png", Wait_time):
        dragDrop("1751877757141.png", Location(1475, 29))
        symbol_moved = 1
    elif exists("1751889709844.png", Wait_time):
        dragDrop("1751889709844.png", Location(1475, 29))
        symbol_moved = 1
    elif exists("1751889775827.png", Wait_time):
        dragDrop("1751889775827.png", Location(1475, 29))
        symbol_moved = 1
    elif exists("1751889831574.png", Wait_time):
        dragDrop("1751889831574.png", Location(1475, 29))
        symbol_moved = 1

def Open_Toolbar():
    if exists("1751877757141.png", Wait_time):
        click("1751877757141.png")
    elif exists("1751889709844.png", Wait_time):
        click("1751889709844.png")
    elif exists("1751889775827.png", Wait_time):
        click("1751889775827.png")
    elif exists("1751889831574.png", Wait_time):
        click("1751889831574.png")

# Close PowerOn toolbar
def Close_Toolbar():
    if exists("1751877851392.png", Wait_time):
        click(Pattern("1751877851392.png").targetOffset(876, -57))
    elif exists("1751890268170.png", Wait_time):
        click(Pattern("1751890268170.png").targetOffset(208, -486))

# Choose functionality from Toolbar
def Choose_Toolbar(mode):
    if mode == "Network_Diagram" or mode == "GeoView":
        click(Pattern("1751890383516.png").similar(0.80))
    if mode == "Network_Diagram":
        click(Pattern("1751877902405.png").similar(0.80))
    if mode == "GeoView":
        click(Pattern("1751992457030.png").similar(0.80))

    if mode == "Work_Package_Manager" or mode == "Trends":
        click(Pattern("1752487267647.png").similar(0.80))
    if mode == "Work_Package_Manager":
        click(Pattern("1752607339428.png").similar(0.80).targetOffset(-16, -21))
    if mode == "Trends":
        click(Pattern("1752607339428.png").similar(0.80).targetOffset(-96, -73))
```

```
if mode == "Outstanding_Calls":
    click(Pattern("1752599041695.png").similar(0.80).targetOffset(-5, 3))
    click(Pattern("1752599169682.png").similar(0.80).targetOffset(-20, -20))

# Unpin PowerOn toolbar
def Unpin_Toolbar():
    if exists(Pattern("1751975384555.png").similar(0.88), Wait_time):
        click("1751975384555.png")

# Check Navigation mode
def CheckNavigationMode():
    if exists(Pattern("1751879503968.png").similar(0.80), Wait_time):
        click(Pattern("1751879503968.png").targetOffset(22, 1))

# Filters check for substation
def FilterForSubstation():
    click(Pattern("1752064695766.png").similar(0.80).targetOffset(37, 1))
    click(Pattern("1751885464921.png").similar(0.83))

if not exists(Pattern("1751967404051.png").similar(0.98), Wait_time3): # Search mode in all
    click(Pattern("1751967404051.png").targetOffset(-39, 64))

if exists(Pattern("1751885700026.png").similar(0.94), Wait_time3): # All filter for location type out
    click(Pattern("1751885792256.png").targetOffset(-143, 128))

if exists(Pattern("1751886544236.png").similar(0.95), Wait_time3): # Primary filter on
    click(Pattern("1751885792256.png").targetOffset(-133, 74))

if exists(Pattern("1751889547180.png").similar(0.95).targetOffset(-2, 1), Wait_time3): # Secondary filter on
    # Secondary filter on (not sure if I have to take this filter out)
    click(Pattern("1751885792256.png").targetOffset(-128, 102))

if exists(Pattern("1751886075205.png").similar(0.95), Wait_time3): # Transmission filter out
    click(Pattern("1751885792256.png").targetOffset(-115, -15))

if exists(Pattern("1751886120028.png").similar(0.95), Wait_time3): # Sub Transmission filter out
    click(Pattern("1751885792256.png").targetOffset(-112, 15))

# if exists(Pattern("1752000601982.png").similar(0.95), Wait_time3): # Grid filter out (Maybe this one)
#     click(Pattern("1751885792256.png").targetOffset(-143, 44))

if exists(Pattern("1752485338597.png").similar(0.95), Wait_time3): # Grid filter on (Maybe this out)
    click(Pattern("1751885792256.png").targetOffset(-143, 44))

if exists(Pattern("1751887647933.png").similar(0.95), Wait_time3): # Only visible locations filter on
    click(Pattern("1751885792256.png").targetOffset(-86, -143))

if exists(Pattern("1751889204156.png").similar(0.95), Wait_time3): # Subtype as substation
    click(Pattern("1751885792256.png").targetOffset(-121, 217))
```

```
if exists(Pattern("1751967781186.png").similar(0.95), Wait_time3): # Case Sensitive filter out
    click(Pattern("1751885792256.png").targetOffset(-107, -175))

# if exists(Pattern("1751968049117.png").similar(0.95), Wait_time3): # LV Location Only filter out (maybe
having this on would be better)
#    click(Pattern("1751885792256.png").targetOffset(-110, -113))

if exists(Pattern("1752484225341.png").similar(0.95), Wait_time3):
    click(Pattern("1751885792256.png").targetOffset(-111, -114))

if exists(Pattern("1751968595446.png").similar(0.95), Wait_time3):
    click(Pattern("1751885792256.png").targetOffset(-66, -82))

def MaximizeScreen():
    if exists(Pattern("1751890593478.png").similar(0.50), Wait_time):
        click(Pattern("1751890593478.png").targetOffset(-2, 3))
    elif exists(Pattern("1753010653117.png").similar(0.80), Wait_time):
        click(Pattern("1753010653117.png").similar(0.80))

def CloseWindow():
    if exists("1752605073193.png", Wait_time):
        click(Pattern("1752605073193.png").targetOffset(47, 2))

def SearchLocation(mode):
    click("1751993013582.png")

    if not exists(Pattern("1751970421874.png").exact(), Wait_time):
        type("a", Key.CTRL)
        type(Key.DELETE)

    # Close_Toolbar(
    wait(0.5)
    type("v", Key.CTRL)

    if exists(Pattern("1752622220776.png").similar(0.35), Wait_time4):
        click(Pattern("1752622220776.png").similar(0.40).targetOffset(-194, -164))
    # type(Key.ENTER) # Causes issues

    if exists(Pattern("1751972271215.png").similar(0.40), Wait_time) and mode == "Network_diagram":
        click(Pattern("1751972271215.png").similar(0.40).targetOffset(4, 54))
        click(Pattern("1751993013582.png").targetOffset(145, -1))
        click(Pattern("1751976333420.png").similar(0.36).targetOffset(-383, -250))

    if exists(Pattern("1751977259495.png").similar(0.50), Wait_time4) and mode == "Network_diagram": # 1 sec
wait is good
        click(Pattern("1751982526876.png").similar(0.90))
        click(Pattern("1751982236827.png").targetOffset(-59, 3))

    # Adjust scale in Network Diagram
    if mode == "Network_diagram":
```

```
click(Pattern("1751992054822.png").similar(0.60).targetOffset(-703, 490))
type("a", Key.CTRL)
type(Key.DELETE)
type("1.6")
# Choose a proper scale depending on requirements

# Adjust scale in GeoView (Doesn't work because mouse is blocked by computer)
# When it is open it is in 2.5 so I assume it is good enough
# if mode == "GeoView":
#   wheel(Pattern("1751998943062.png").similar(0.50).targetOffset(-5, 6), WHEEL_UP, 30)
#   wheel(Pattern("1751998943062.png").similar(0.50), WHEEL_DOWN, 11) # I set it to 1.6

def SearchNameWPM():
    wait(0.1) # To make sure the code runs correctly

    if not exists(Pattern("1752490547688.png").targetOffset(-87, 2), Wait_time4):
        if exists(Pattern("1752489543538.png").similar(0.90), Wait_time2):
            rightClick(Pattern("1752489543538.png").similar(0.90))

        if exists("1753048423594.png", Wait_time):
            click(Pattern("1753048423594.png").targetOffset(-39, 42))
            elif exists("1753090994837.png", Wait_time): # this line is just for schedule as SikuliX does not recognize
the upper line (I don't know why)
                click(Pattern("1753090994837.png").targetOffset(-38, 27))

    if exists(Pattern("1752490547688.png").targetOffset(-87, 2), Wait_time4):
        click(Pattern("1752490547688.png").targetOffset(-87, 2))
        type("a", Key.CTRL)
        type(Key.DELETE)
        wait(0.5)
        type("v", Key.CTRL)

    waitVanish(Pattern("1753091860079.png").similar(0.40), Wait_time5)
    waitVanish(Pattern("1753088998471.png").similar(0.40), Wait_time5)

    doubleClick(Pattern("1753048618278.png").similar(0.30).targetOffset(-36, -127))

def IsToolbarClosed():
    if not exists("1751877851392.png", Wait_time) or not exists("1751890268170.png", Wait_time):
        return True
    else:
        return False

def GeoViewTests(mode):
    global GeoViewScaleChanged

    if GeoViewScaleChanged == 0:
        popup("Please first adjust GeoView Scale to 2.000")
        GeoViewScaleChanged = 1
```

```
if mode == 1:
    GeoViewOpened()
    substation_GV = input("Please enter the name of the substation you want to search for:", "", "Substation Search")

    # Check that the user entered something
    if not substation_GV:
        popup("No substation name entered. The test will now be finalized.")

    copy_to_clipboard(substation_GV)

    if substation_GV:
        FilterForSubstation()
        SearchLocation("GeoView")
        popup("Substation found in GeoView")

if mode == 2: # Go to incident / View incident
    GeoViewOpened()
    incident_GV = input("Please enter where you want to search for an incident:", "", "Incident Search")

    if not incident_GV:
        popup("No incident entered. The test will now be finalized.")

    if incident_GV:
        copy_to_clipboard(incident_GV)
        FilterForSubstation()
        SearchLocation("GeoView")
        rightClick(Pattern("1752617934947.png").similar(0.40).targetOffset(-14, 90))
        click(Pattern("1752618070595.png").targetOffset(-67, 56))

        if exists(Pattern("1752618140671.png").similar(0.50), Wait_time2):
            click(Pattern("1752618140671.png").similar(0.50).targetOffset(0, 56))
            popup("There are no incidents in this location")
        elif not exists(Pattern("1752618140671.png").similar(0.50), Wait_time4):
            if exists(Pattern("1751992054822.png").similar(0.35).targetOffset(-703, 490), Wait_time4):
                click(Pattern("1751992054822.png").similar(0.60).targetOffset(-703, 490))
                type("a", Key.CTRL)
                type(Key.DELETE)
                type("5")

            if exists(Pattern("1752618698752.png").similar(0.45), Wait_time4):
                rightClick(Pattern("1752618698752.png").similar(0.45))
                click(Pattern("1752618752308.png").targetOffset(-9, 189))
                MaximizeScreen()

        popup("These are all the incidents in this location")

    popup("Incidents search test finalized")

if mode == 3: # Display Customers
    GeoViewOpened()
    customers_GV = input("Please enter the substation for customers display:", "", "Customers Search")
```

```

if not customers_GV:
    popup("No substation entered. The test will now be finalized.")

if customers_GV:
    copy_to_clipboard(customers_GV)
    FilterForSubstation()
    SearchLocation("GeoView")
    rightClick(Pattern("1752617934947.png").similar(0.40).targetOffset(-14, 90))
    click(Pattern("1752618070595.png").targetOffset(-53, -9))

    if exists(Pattern("1752619765192.png").similar(0.30), Wait_time4):
        popup("These are the customers from the selected substation. Press OK when you want to finish the
test")
        click(Pattern("1752619765192.png").similar(0.30).targetOffset(616, -448))

        popup("Display customers test finalized")

if mode == 4: # Position substation on HV diagram
    GeoViewOpened()
    substation2_GV = input("Please enter the name of the substation you want to search for:", "", "Substation
Search")

    # Check that the user entered something
    if not substation2_GV:
        popup("No substation name entered. The test will now be finalized.")

    if substation2_GV:
        copy_to_clipboard(substation2_GV)
        FilterForSubstation()
        SearchLocation("GeoView")
        rightClick(Pattern("1752617934947.png").similar(0.40).targetOffset(-14, 90))
        click(Pattern("1752618070595.png").targetOffset(-71, 21))
        popup("This is the position of the substation entered in the HV diagram. Press OK when you want to finish
the test")
        CloseWindow()

        popup("Substation position on HV diagram test finalized")

def OpenWorkPackageManager(mode):
    # Modes:
    # 1 Open Faults
    # 2 Open Jobs
    # 3 Open Schedules
    # 4 Permit for Work
    # 5 Sanction for Test
    # 6 General Consent
    # 7 OR
    # 8 SOP
    # 9 Technical Limitation Record
    # This is only for searching faults, jobs or schedules
    global WPMMovedCheck

```

WorkPackageManagerOpened()

if mode == 1 or mode == 2 or mode == 3:

if exists(Pattern("1752624681538.png").exact(), Wait_time):

click(Pattern("1752488390243.png").targetOffset(-85, 2))

elif exists(Pattern("1752624641514.png").exact(), Wait_time):

click(Pattern("1752488437783.png").targetOffset(-81, 2))

elif exists(Pattern("1752624708518.png").exact(), Wait_time):

click(Pattern("1752488465519.png").targetOffset(-85, 3))

if WPMMovedCheck == 0:

dragDrop(Pattern("1752491434911.png").similar(0.90).targetOffset(1, -21), Location(1012, 585))

WPMMovedCheck = 1

if mode == 1:

Ask where the Fault is being searched

Fault = input("Please enter where the Fault is:", "", "Fault Search")

copy_to_clipboard(Fault)

Check that the user entered something

if not Fault:

popup("No place name entered. Fault test ending.")

if Fault:

if exists(Pattern("1752491115077.png").exact(), Wait_time):

click(Pattern("1752491115077.png").exact())

wait(Pattern("1753090068305.png").similar(0.65), Wait_time5)

SearchNameWPM()

popup("Fault test completed")

elif mode == 2:

Ask where the Job is being searched

Job = input("Please enter where the Job is:", "", "Job Search")

copy_to_clipboard(Job)

Check that the user entered something

if not Job:

popup("No place name entered. Job test ending.")

if Job:

if exists(Pattern("1752488751438.png").similar(0.90), Wait_time):

click(Pattern("1752488751438.png").similar(0.90))

wait(Pattern("1753090087616.png").similar(0.65), Wait_time5)

SearchNameWPM()

popup("Job test completed")

elif mode == 3:

Ask where the Schedule is being searched

Schedule = input("Please enter where the Schedule is:", "", "Schedule Search")

copy_to_clipboard(Schedule)

```
# Check that the user entered something
if not Schedule:
    popup("No place name entered. Schedule test ending.")

if Schedule:
    if exists(Pattern("1753090595025.png").exact(), Wait_time):
        click(Pattern("1753090595025.png").exact())

    wait(Pattern("1753089939464.png").similar(0.65), Wait_time5)
    SearchNameWPM()
    popup("Schedule test completed")

# This is only for Safety Documents
if mode >= 4 and mode <= 9:
    if mode == 4 or mode == 5 or mode == 6: # modes 4, 5 and 6 are the same except at the end
        copy_to_clipboard("Type")
        click(Pattern("1752581747421.png").similar(0.85).targetOffset(2, 4)) # Go to find documents
        click(Pattern("1752581870462.png").similar(0.80).targetOffset(852, -39))

    if exists("1752581919443.png", Wait_time):
        click(Pattern("1752581919443.png").similar(0.75).targetOffset(-85, 50))

    click(Pattern("1752582020875.png").similar(0.90).targetOffset(-697, 11))
    type("a", Key.CTRL)
    type(Key.DELETE)
    wait(0.5)
    type("v", Key.CTRL)
    type(Key.ENTER)
    click(Pattern("1752582020875.png").similar(0.90).targetOffset(322, 20))

    # So it starts searching from the top
    keyDown(Key.UP)
    wait(Pattern("1752582634299.png").similar(0.95), Wait_time5)
    keyUp(Key.UP)

    if mode == 4: # PFW
        PFW = Pattern("1752583265800.png").similar(0.95)
        run_for_duration(Wait_time7, PressDownWhile, PFW)

        if exists(Pattern("1752583265800.png").similar(0.95), Wait_time):
            click(Pattern("1752583265800.png").similar(0.95))
            click(Pattern("1752581870462.png").similar(0.60).targetOffset(814, 255))
            click(Pattern("1752581870462.png").similar(0.60).targetOffset(837, -127))
            dragDrop(Pattern("1752581870462.png").targetOffset(-49, -250), Location(927, 122))
            wait(Pattern("1752583784004.png").similar(0.40), Wait_time6) # It takes a lot of time
            popup("All Permits for Work found. Press OK when you finish to continue.")
            # This part can be changed if there is a better idea
        else:
            popup("ERROR. Mode not found")

    click(Pattern("1752581870462.png").similar(0.60).targetOffset(931, -258)) # Close

if mode == 5: # SFT
```

```
SFT = Pattern("1752584245343.png").similar(0.95).targetOffset(-3, 3)
run_for_duration(Wait_time7, PressDownWhile, SFT)

if exists(Pattern("1752584245343.png").similar(0.95).targetOffset(-3, 3), Wait_time):
    click(Pattern("1752584245343.png").similar(0.95).targetOffset(-3, 3))
    click(Pattern("1752581870462.png").similar(0.60).targetOffset(814, 255))
    click(Pattern("1752581870462.png").similar(0.60).targetOffset(837, -127))
    wait(Pattern("1752583784004.png").similar(0.40), Wait_time6) # It takes a lot of time
    popup("All Sanction for Test documents found. Press OK when you finish to continue.")
    # This part can be changed if there is a better idea
else:
    popup("ERROR. Mode not found")

click(Pattern("1752581870462.png").similar(0.60).targetOffset(931, -258)) # Close

if mode == 6: # GenCon
    GenCon = Pattern("1752585796011.png").similar(0.95)
    run_for_duration(Wait_time7, PressDownWhile, GenCon)

    if exists(Pattern("1752585796011.png").similar(0.95), Wait_time):
        click(Pattern("1752585796011.png").similar(0.95))
        click(Pattern("1752581870462.png").similar(0.60).targetOffset(814, 255))
        click(Pattern("1752581870462.png").similar(0.60).targetOffset(837, -127))
        wait(Pattern("1752583784004.png").similar(0.40), Wait_time6) # It takes a lot of time
        popup("All General Consent documents found. Press OK when you finish to continue.")
        # This part can be changed if there is a better idea
    else:
        popup("ERROR. Mode not found")

    click(Pattern("1752581870462.png").similar(0.60).targetOffset(931, -258)) # Close

if mode == 7 or mode == 8 or mode == 9: # For this I can create a folder too if they want
    copy_to_clipboard("Order Type")
    click(Pattern("1752581747421.png").similar(0.85).targetOffset(79, 3)) # Go to find operations
    click(Pattern("1752586283162.png").similar(0.80))

    if exists("1752586347225.png", Wait_time):
        click(Pattern("1752586347225.png").similar(0.75).targetOffset(-85, 53))

    click(Pattern("1752582020875.png").similar(0.90).targetOffset(-697, 11))
    type("a", Key.CTRL)
    type(Key.DELETE)
    wait(0.5)
    type("v", Key.CTRL)
    type(Key.ENTER)

    click(Pattern("1752582020875.png").similar(0.90).targetOffset(322, 20))

    # So it starts searching from the top
    keyDown(Key.UP)
    wait(Pattern("1752582634299.png").similar(0.95), Wait_time5)
    keyUp(Key.UP)
```

```

if mode == 7: # OR/SOP
    OR = Pattern("OR.png").similar(0.85)
    run_for_duration(Wait_time7, PressDownWhile, OR)

    if exists(Pattern("OR.png").similar(0.85), Wait_time):
        click(Pattern("OR.png").similar(0.85))
        click(Pattern("1752587003505.png").similar(0.60).targetOffset(836, 257))
        click(Pattern("1752587003505.png").similar(0.60).targetOffset(838, -131))
        wait(Pattern("1752583784004.png").similar(0.40), Wait_time6) # It takes a lot of time
        popup("All SOP/OR documents found. Press OK when you finish to continue.")
        # This part can be changed if there is a better idea
    else:
        popup("ERROR. Mode not found")

    click(Pattern("1752587003505.png").similar(0.60).targetOffset(928, -258)) # Close

if mode == 8: # OR/SOP (same as mode 7)
    SOP = Pattern("OR.png").similar(0.85)
    run_for_duration(Wait_time7, PressDownWhile, SOP)

    while not exists(Pattern("OR.png").similar(0.85), 0) or not
exists(Pattern("1753101953717.png").exact(), Wait_time7):
        type(Key.DOWN)
        wait(Wait_time)

    if exists(Pattern("OR.png").similar(0.85), Wait_time):
        click(Pattern("OR.png").similar(0.85))
        click(Pattern("1752587003505.png").similar(0.60).targetOffset(836, 257))
        click(Pattern("1752587003505.png").similar(0.60).targetOffset(838, -131))
        wait(Pattern("1752583784004.png").similar(0.40), Wait_time6) # It takes a lot of time
        popup("All SOP/OR documents found. Press OK when you finish to continue.")
        # This part can be changed if there is a better idea
    else:
        popup("ERROR. Mode not found")

    click(Pattern("1752587003505.png").similar(0.60).targetOffset(928, -258)) # Close

if mode == 9: # TLR Test
    TLR = Pattern("1752587538697.png").similar(0.95)
    run_for_duration(Wait_time7, PressDownWhile, TLR)

    if exists(Pattern("1752587538697.png").similar(0.95), Wait_time):
        click(Pattern("1752587538697.png").similar(0.95))
        click(Pattern("1752587003505.png").similar(0.60).targetOffset(836, 257))
        click(Pattern("1752587003505.png").similar(0.60).targetOffset(838, -131))
        wait(Pattern("1753113488206.png").similar(0.45), Wait_time6) # It takes a lot of time
        popup("All TLR documents found. Press OK when you finish to continue.")
        # This part can be changed if there is a better idea
    else:
        popup("ERROR. Mode not found")

    click(Pattern("1752587003505.png").similar(0.60).targetOffset(928, -258)) # Close

```

APPS CHECKS TO OPEN

Check if GeoView is open or not and open it if necessary

```
def GeoViewOpened():
    if not exists(Pattern("1752619159293.png").similar(0.35), Wait_time):
        Open_Toolbar()
        Choose_Toolbar("GeoView")
        waitVanish("1752064071467.png", Wait_time5)
        Close_Toolbar()
        MaximizeScreen()
```

def NetworkDiagramOpened():

```
    if not exists(Pattern("1753040718915.png").similar(0.35), Wait_time):
        Open_Toolbar()
        Choose_Toolbar("Network_Diagram")
        Close_Toolbar()
        MaximizeScreen()
```

def WorkPackageManagerOpened():

```
    if not exists("1753088346194.png", Wait_time):
        Open_Toolbar()
        Choose_Toolbar("Work_Package_Manager")
        Close_Toolbar()
        MaximizeScreen()
```

def OMSOpened():

```
    if not exists("1753129824252.png", Wait_time):
        Open_Toolbar()
        Choose_Toolbar("Outstanding_Calls")
        Close_Toolbar()
        MaximizeScreen()
```

FUNCTIONS FOR BOX SELECTION

Select which tests the user wants to run

```
def TestSelection(): # I'll add it at the end
    # Set variables to 0
    Test1 = 0 # Network Diagram Test
    Test2 = 0 # GeoView Test
    Test3 = 0 # Work Package Manager Test
    Test4 = 0 # Trace Test
    Test5 = 0 # Safety Documents Test
    Test6 = 0 # PowerOn OMS Test
    Test7 = 0 # PC Call Supervisor Test
    Test8 = 0 # CallTaker Test
    Test9 = 0 # Trends Test
    Test10 = 0 # Alarms Test
```

```
# Create a JFrame window
frame = JFrame("Select Tests")
frame.setSize(1000, 500)
frame.setLayout(GridLayout(0, 2))

# Create checkboxes
cb1 = JCheckBox("Network Diagram Test")
cb2 = JCheckBox("GeoView Test")
cb3 = JCheckBox("Work Package Manager Test")
cb4 = JCheckBox("Trace Test")
cb5 = JCheckBox("Safety Documents Test")
cb6 = JCheckBox("PowerOn OMS Test")
cb7 = JCheckBox("PC Call Supervisor Test")
cb8 = JCheckBox("CallTaker Test")
cb9 = JCheckBox("Trends Test")
cb10 = JCheckBox("Alarms Test")

# Add checkboxes to frame
frame.add(cb1)
frame.add(cb2)
frame.add(cb3)
frame.add(cb4)
frame.add(cb5)
frame.add(cb6)
frame.add(cb7)
frame.add(cb8)
frame.add(cb9)
frame.add(cb10)

# Define OK button action
def on_ok(event):
    frame.dispose()

ok_button = JButton("OK", actionPerformed=on_ok)
frame.add(ok_button)

# Show the window
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE)
frame.setLocationRelativeTo(None) # Center the box
frame.setVisible(True)

# Wait for window to close
while frame.isVisible():
    wait(0.2)

# Check which tests were selected
if (cb1.isSelected() or cb2.isSelected() or cb3.isSelected() or cb4.isSelected() or
    cb5.isSelected() or cb6.isSelected() or cb7.isSelected() or cb8.isSelected() or
    cb9.isSelected() or cb10.isSelected()):
    popup("Running Tests")
else:
    popup("No Test selected. Finalizing the program")
    exit()
```

```
if cb1.isSelected():
    Test1 = 1
if cb2.isSelected():
    Test2 = 1
if cb3.isSelected():
    Test3 = 1
if cb4.isSelected():
    Test4 = 1
if cb5.isSelected():
    Test5 = 1
if cb6.isSelected():
    Test6 = 1
if cb7.isSelected():
    Test7 = 1
if cb8.isSelected():
    Test8 = 1
if cb9.isSelected():
    Test9 = 1
if cb10.isSelected():
    Test10 = 1
```

```
return Test1, Test2, Test3, Test4, Test5, Test6, Test7, Test8, Test9, Test10
```

```
# Select GeoView Tests
def SelectionGeoView():
    GeoViewTest1 = 0
    GeoViewTest2 = 0
    GeoViewTest3 = 0
    GeoViewTest4 = 0
    SelectedGeoViewTest = 1

    frame = JFrame("Select Tests")
    frame.setSize(1000, 500)
    frame.setLayout(GridLayout(0, 1))

    # Create checkboxes
    cb1 = JCheckBox("Search for Substation")
    cb2 = JCheckBox("Search for Incident")
    cb3 = JCheckBox("Display Customers")
    cb4 = JCheckBox("Position Substation on HV diagram")

    # Add checkboxes to frame
    frame.add(cb1)
    frame.add(cb2)
    frame.add(cb3)
    frame.add(cb4)

    # Define OK button action
    def on_ok(event):
        frame.dispose()
```

```
ok_button = JButton("OK", actionPerformed=on_ok)
frame.add(ok_button)

# Show the window
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE)
frame.setLocationRelativeTo(None) # Center the box
frame.setVisible(True)

# Wait for window to close
while frame.isVisible():
    wait(0.2)

# Check which tests were selected
if cb1.isSelected() or cb2.isSelected() or cb3.isSelected() or cb4.isSelected():
    popup("Running Tests")
else:
    popup("No Test selected. Finalizing the GeoView Test")
    SelectedGeoViewTest = 0

if cb1.isSelected():
    GeoViewTest1 = 1
if cb2.isSelected():
    GeoViewTest2 = 1
if cb3.isSelected():
    GeoViewTest3 = 1
if cb4.isSelected():
    GeoViewTest4 = 1

return GeoViewTest1, GeoViewTest2, GeoViewTest3, GeoViewTest4, SelectedGeoViewTest

# Select Work Package Manager Tests
def SelectionWorkPackageManager():
    # Set variables to 0
    Fault = 0
    Job = 0
    Schedule = 0
    SelectedWPMTest = 1

    # Create a JFrame window
    frame = JFrame("Select Work Package Manager Tests")
    frame.setSize(1000, 500)
    frame.setLayout(GridLayout(0, 1))

    # Create checkboxes
    cb1 = JCheckBox("Open Fault Test")
    cb2 = JCheckBox("Open Job Test")
    cb3 = JCheckBox("Open Schedule Test")

    # Add checkboxes to frame
    frame.add(cb1)
    frame.add(cb2)
    frame.add(cb3)
```

```
# Define OK button action
def on_ok1(event):
    frame.dispose()

ok_button = JButton("OK", actionPerformed=on_ok1)
frame.add(ok_button)

# Show the window
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE)
frame.setLocationRelativeTo(None) # Center the box
frame.setVisible(True)

# Wait for window to close
while frame.isVisible():
    wait(0.2)

# Check which tests were selected
if cb1.isSelected() or cb2.isSelected() or cb3.isSelected():
    popup("Running Selected Work Package Manager Test")
else:
    popup("No test selected. Finalizing the test")
    SelectedWPMTest = 0

if cb1.isSelected():
    Fault = 1
if cb2.isSelected():
    Job = 1
if cb3.isSelected():
    Schedule = 1

return Fault, Job, Schedule, SelectedWPMTest

# Select which Safety Documents the user wants
def SelectionSafetyDocuments():
    # Set variables to 0
    PFW = 0
    SFT = 0
    GenConsent = 0
    SOP = 0
    OR = 0
    TLR = 0
    SelectedSafetyDocuments = 1

# Create a JFrame window
frame = JFrame("Select Safety Documents")
frame.setSize(1000, 500)
frame.setLayout(GridLayout(0, 2))

# Create checkboxes
cb1 = JCheckBox("Permit For Work (PFW)")
cb2 = JCheckBox("Sanction For Test (SFT) (Not found in Work Package Manager)")
```

```
cb3 = JCheckBox("General Consent (Not found in Work Package Manager)")
cb4 = JCheckBox("SOP")
cb5 = JCheckBox("Operation Restriction (OR)")
cb6 = JCheckBox("Technical Limitation Record (TLR)")

# Add checkboxes to frame
frame.add(cb1)
frame.add(cb2)
frame.add(cb3)
frame.add(cb4)
frame.add(cb5)
frame.add(cb6)

# Define OK button action
def on_ok1(event):
    frame.dispose()

ok_button = JButton("OK", actionPerformed=on_ok1)
frame.add(ok_button)

# Show the window
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE)
frame.setLocationRelativeTo(None) # Center the box
frame.setVisible(True)

# Wait for window to close
while frame.isVisible():
    wait(0.2)

# Check which tests were selected
if cb1.isSelected() or cb2.isSelected() or cb3.isSelected() or cb4.isSelected() or cb5.isSelected() or
cb6.isSelected():
    popup("Running Selected Safety Documents Test")
else:
    popup("No documents selected. Finalizing the test")
    SelectedSafetyDocuments = 0

if cb1.isSelected():
    PFW = 1
if cb2.isSelected():
    SFT = 1
if cb3.isSelected():
    GenConsent = 1
if cb4.isSelected():
    SOP = 1
if cb5.isSelected():
    OR = 1
if cb6.isSelected():
    TLR = 1

return PFW, SFT, GenConsent, SOP, OR, TLR, SelectedSafetyDocuments
```

```
# MAIN
# Move symbol to a good place and unpin Toolbar

Open_Toolbar()
# Unpin_Toolbar()
Close_Toolbar()
Move_Symbol()

# Select the tests to run
Test1, Test2, Test3, Test4, Test5, Test6, Test7, Test8, Test9, Test10 = TestSelection()

# Network Diagram test
if Test1 == 1:
    # Open the diagram tool and search
    NetworkDiagramOpened()

    # Checking Navigation mode
    CheckNavigationMode()

    # Prompt the user to input the substation name they want to search for
    substation_ND = input("Please enter the name of the substation you want to search for:", "", "Substation
Search")

    # Check that the user entered something
    if not substation_ND:
        popup("No substation name entered. The script will now exit.")
        exit()

    if substation_ND:
        # Copy the substation
        copy_to_clipboard(substation_ND)
        # Search for substation
        FilterForSubstation()
        SearchLocation("Network_diagram")
        popup("Substation found in Network Diagram") # show as popup

    CloseWindow()
    popup("Network Diagram Tests finalized")

# GeoView Test
if Test2 == 1:
    GeoViewTest1, GeoViewTest2, GeoViewTest3, GeoViewTest4, SelectedGeoViewTest = SelectionGeoView()

    if SelectedGeoViewTest == 1:
        # For any test (already placed in the function)
        GeoViewOpened() # just open if it is not

        if GeoViewTest1 == 1:
            GeoViewMode = 1
            GeoViewTests(GeoViewMode)

        if GeoViewTest2 == 1:
            GeoViewMode = 2
```

```
GeoViewTests(GeoViewMode)

if GeoViewTest3 == 1:
    GeoViewMode = 3
    GeoViewTests(GeoViewMode)

if GeoViewTest4 == 1:
    GeoViewMode = 4
    GeoViewTests(GeoViewMode)

CloseWindow()
popup("GeoView Tests finalized")

# Work Package Manager Test
if Test3 == 1:
    WPMTest1, WPMTest2, WPMTest3, SelectedWPMTest = SelectionWorkPackageManager()

    if SelectedWPMTest == 1:
        WorkPackageManagerOpened()

    if WPMTest1 == 1: # Fault Test
        WorkPackageManager_mode = 1
        OpenWorkPackageManager(WorkPackageManager_mode)

    if WPMTest2 == 1: # Job Test
        WorkPackageManager_mode = 2
        OpenWorkPackageManager(WorkPackageManager_mode)

    if WPMTest3 == 1: # Schedule Test
        WorkPackageManager_mode = 3
        OpenWorkPackageManager(WorkPackageManager_mode)

CloseWindow()
popup("Work Package Manager Test finalized")

# TRACE (need an explanation)
# if Test4 == 1:

# Safety Documents Test
if Test5 == 1:
    # Checkbox selecting which documents the operator needs, the program will create a new folder for each type
    # of document (PFW, SFT, SOP/OR, TLR)
    PFW, SFT, GenConsent, SOP, OR, TLR, SelectedSafetyDocuments = SelectionSafetyDocuments()

    if SelectedSafetyDocuments == 1:
        WorkPackageManagerOpened()

if SelectedSafetyDocuments == 1: # Only run this if there is a Document selected
    if PFW == 1:
        WorkPackageManager_mode = 4
        OpenWorkPackageManager(WorkPackageManager_mode)

    if SFT == 1:
```

```
WorkPackageManager_mode = 5
OpenWorkPackageManager(WorkPackageManager_mode)

if GenConsent == 1:
    WorkPackageManager_mode = 6
    OpenWorkPackageManager(WorkPackageManager_mode)

if SOP == 1:
    WorkPackageManager_mode = 7
    OpenWorkPackageManager(WorkPackageManager_mode)

if OR == 1:
    WorkPackageManager_mode = 8
    OpenWorkPackageManager(WorkPackageManager_mode)

if TLR == 1:
    WorkPackageManager_mode = 9
    OpenWorkPackageManager(WorkPackageManager_mode)

CloseWindow()
popup("Safety Documents Test finalized")

# PowerOn OMS test
if Test6 == 1:
    OMSSOpened

# All operating zones available
wait("1752657016143.png", Wait_time2)

if exists(Pattern("1752601436067.png").similar(0.95), Wait_time3):
    click(Pattern("1752601436067.png").similar(0.95).targetOffset(208, 3))

# These can be changed to select which zones the user wants
click(Pattern("1752601568445.png").exact().targetOffset(-100, 1))

# Just checking if it works with the first call
wait(Pattern("1752601839879.png").similar(0.35), Wait_time)
doubleClick(Pattern("1752601839879.png").similar(0.35).targetOffset(-127, -358)) # First call
wait(Pattern("1752601964370.png").similar(0.40), Wait_time2)
popup("Here is the call")

# Go to Log Message
click(Pattern("1752602015511.png").similar(0.95))
popup("Here is the incident log")

# Position on diagram
Positions = ["Diagram Network", "GeoView", "Call"]
Position = select("Now the call is going to be positioned. Please choose how you want to position the call:",
"Position selection", Positions, Positions[0])

if not Position:
    popup("No position selected. Going to next step")
else:
```

```

click(Pattern("1752602229793.png").similar(0.95).targetOffset(49, 2))

if Position == Positions[0]:
    click(Pattern("1752603423915.png").similar(0.90).targetOffset(-19, -33))
    wait(Pattern("1752603797536.png").similar(0.40), Wait_time2)
    popup("Call positioned correctly")
    CloseWindow()

if Position == Positions[1]:
    click(Pattern("1752603423915.png").similar(0.90).targetOffset(-20, 1))
    waitVanish("1752064071467.png", Wait_time5)
    popup("Call positioned correctly")
    CloseWindow()

if Position == Positions[2]:
    click(Pattern("1752603423915.png").similar(0.90).targetOffset(-55, 40))
    # wait(, Wait_time2)
    popup("Call positioned correctly")
    CloseWindow()

popup("PowerOn OMS Test finalized")

# PC Call Supervisor Test
# if Test7 == 1: don't know how this works

# CallTaker Test
# if Test8 == 1: don't know how this works

# Trends Test
if Test9 == 1:
    if IsToolbarClosed():
        OpenToolbar()
        Choose_Toolbar(Trends)

    if not IsToolbarClosed():
        Close_Toolbar()

# Can't do much more with the info
popup("Trends Test finalized")

# Alarms Test
# if Test10 == 1:

```

12 Appendix C: Batch Tests Code

```

import csv, os, time
from javax.swing import JFrame, JCheckBox, JButton
from java.awt import GridLayout
from java.awt.datatransfer import StringSelection
from java.awt import Toolkit
#from org.sikuli.script import FindFailed, Pattern, Screen, Location, Settings

# =====

```

ERRORS / USEFUL FUNTIONS

=====

```
class TestError(Exception):
    def __init__(self, step, msg, screenshot=None):
        self.step = step
        self.screenshot = screenshot
        super(TestError, self).__init__("[%s] %s" % (step, msg))

def snap(label="fail"):
    ts = time.strftime("%Y%m%d_%H%M%S")
    folder = RESULTS_FOLDER if 'RESULTS_FOLDER' in globals() else os.getcwd()
    path = os.path.join(folder, "snap_%s_%s.png" % (label, ts))
    try:
        img = SCREEN.capture(SCREEN)
        img.save(path)
    except:
        pass
    return path

def do(step_name, fn):
    try:
        return fn()
    except TestError:
        raise
    except Exception as e:
        shot = snap(step_name)
        raise TestError(step_name, str(e), screenshot=shot)

def notify(msg):
    if INTERACTIVE:
        popup(msg)
    else:
        print("[INFO]", msg)

# safe wrappers
def safe_wait(img, timeout, step):
    if not exists(img, timeout):
        raise TestError(step, "Timeout waiting for %s" % img)
    return True

def safe_click(img, timeout, step, target=None, similar=None):
    patt = Pattern(img)
    if similar is not None:
        patt = patt.similar(similar)
    if exists(patt, timeout):
        if target:
            click(patt.targetOffset(target[0], target[1]))
        else:
            click(patt)
    else:
        raise TestError(step, "Cannot click; not found: %s" % img)

def safe_type(keys_or_text, step, modifier=None):
    try:
        if modifier:
```

```

        type(keys_or_text, modifier)
    else:
        type(keys_or_text)
    except Exception as e:
        raise TestError(step, "Typing failed: %s" % str(e))

def require_exists(img, timeout, step, similar=None):
    patt = Pattern(img)
    if similar is not None:
        patt = patt.similar(similar)
    if not exists(patt, timeout):
        raise TestError(step, "Post-condition failed: %s not visible" % img)

def with_timeout(seconds, step, fn):
    start = time.time()
    last_exc = None
    while time.time() - start < seconds:
        try:
            return fn()
        except FindFailed as e:
            last_exc = e
            wait(0.2)
        except TestError:
            raise
        except Exception as e:
            last_exc = e
            wait(0.2)
    raise TestError(step, "Timeout after %ss (%s)" % (seconds, str(last_exc or "")))

# =====
# GLOBAL VARIABLES
# =====

# Timings
Wait_time = 0.5
Wait_time2 = 2
Wait_time3 = 0.01
Wait_time4 = 1
Wait_time5 = 10
Wait_time6 = 120
Wait_time7 = 30

# States
symbol_moved = 0
GeoViewScaleChanged = 0
WPMMovedCheck = 0

# Paths (usar raw strings)
SUBSTATION_LIST = r"C:\Users\B620717\Desktop\PowerOnData\SubstationDataset.csv"
RESULTS_FOLDER = r"C:\Users\B620717\Desktop\PowerOnResults"

# Modo UI
INTERACTIVE = True # It adjust in MAIN dependint on Single/Batch
SCREEN = Screen()

# =====

```

```
# SUPPORT FUNCTIONS
# =====

def copy_to_clipboard(text):
    clipboard = Toolkit.getDefaultToolkit().getSystemClipboard()
    selection = StringSelection(text)
    clipboard.setContents(selection, None)

def run_for_duration(duration_secs, action, stop_if_found=None):
    start_time = time.time()
    while time.time() - start_time < duration_secs:
        if stop_if_found and exists(stop_if_found, 0):
            break
        action()
        wait(0.2)

def PressDownWhile():
    type(Key.DOWN)

# =====
# POWERON UI HELPERS
# =====

def Move_Symbol():
    global symbol_moved
    for img in
["1751877757141.png", "1751889709844.png", "1751889775827.png", "1751889831574.png"]:
        if exists(img, Wait_time):
            dragDrop(img, Location(1475, 29))
            symbol_moved = 1
            return

def Open_Toolbar():
    for img in
["1751877757141.png", "1751889709844.png", "1751889775827.png", "1751889831574.png"]:
        if exists(img, Wait_time):
            safe_click(img, Wait_time, "Open Toolbar", target =(37,1))
            return

def Close_Toolbar():
    if exists("1751877851392.png", Wait_time):
        click(Pattern("1751877851392.png").targetOffset(876,-57))
    elif exists("1751890268170.png", Wait_time):
        click(Pattern("1751890268170.png").targetOffset(208,-486))

def Choose_Toolbar(mode):
    if mode in ("Network_Diagram", "GeoView"):
        click(Pattern("1751890383516.png").similar(0.80))
        if mode == "Network_Diagram":
            click(Pattern("1751877902405.png").similar(0.80))
        else:
            click(Pattern("1751992457030.png").similar(0.80))
    elif mode in ("Work_Package_Manager", "Trends"):
        click(Pattern("1752487267647.png").similar(0.80))
        if mode == "Work_Package_Manager":
            click(Pattern("1752607339428.png").similar(0.80).targetOffset(-16,-21))
```

```
else:
    click(Pattern("1752607339428.png").similar(0.80).targetOffset(-96,-73))
elif mode == "Outstanding_Calls":
    click(Pattern("1752599041695.png").similar(0.80).targetOffset(-5,3))
    click(Pattern("1752599169682.png").similar(0.80).targetOffset(-20,-20))

def Unpin_Toolbar():
    if exists(Pattern("1751975384555.png").similar(0.88), Wait_time):
        click("1751975384555.png")

def CheckNavigationMode():
    if exists(Pattern("1751879503968.png").similar(0.80), Wait_time):
        click(Pattern("1751879503968.png").targetOffset(22,1))

def FilterForSubstation():
    # Abre filtro
    safe_click("1752064695766.png", Wait_time2, "Open filters", target=(37,1))
    # All scope
    if not exists(Pattern("1751967404051.png").similar(0.98), Wait_time3):
        click(Pattern("1751967404051.png").targetOffset(-39,64))
    # Ajustes de filtros (tal cual tu lógica actual)
    if exists(Pattern("1751885700026.png").similar(0.94), Wait_time3):
        click(Pattern("1751885792256.png").targetOffset(-143,128))
    if exists(Pattern("1751886544236.png").similar(0.95), Wait_time3):
        click(Pattern("1751885792256.png").targetOffset(-133,74))
    if exists(Pattern("1751889547180.png").similar(0.95).targetOffset(-2,1), Wait_time3):
        click(Pattern("1751885792256.png").targetOffset(-128,102))
    if exists(Pattern("1751886075205.png").similar(0.95), Wait_time3):
        click(Pattern("1751885792256.png").targetOffset(-115,-15))
    if exists(Pattern("1751886120028.png").similar(0.95), Wait_time3):
        click(Pattern("1751885792256.png").targetOffset(-112,15))
    if exists(Pattern("1752485338597.png").similar(0.95), Wait_time3): # Grid on/off
        click(Pattern("1751885792256.png").targetOffset(-143,44))
    if exists(Pattern("1751887647933.png").similar(0.95), Wait_time3):
        click(Pattern("1751885792256.png").targetOffset(-86,-143))
    if exists(Pattern("1751889204156.png").similar(0.95), Wait_time3):
        click(Pattern("1751885792256.png").targetOffset(-121,217))
    if exists(Pattern("1751967781186.png").similar(0.95), Wait_time3):
        click(Pattern("1751885792256.png").targetOffset(-107,-175))
    if exists(Pattern("1752484225341.png").similar(0.95), Wait_time3):
        click(Pattern("1751885792256.png").targetOffset(-111,-114))
    if exists(Pattern("1751968595446.png").similar(0.95), Wait_time3):
        click(Pattern("1751885792256.png").targetOffset(-66,-82))

def MaximizeScreen():
    if exists(Pattern("1751890593478.png").similar(0.50), Wait_time):
        click(Pattern("1751890593478.png").targetOffset(-2,3))
    elif exists(Pattern("1753010653117.png").similar(0.80), Wait_time):
        click(Pattern("1753010653117.png").similar(0.80))

def CloseWindow():
    if exists("1752605073193.png", Wait_time):
        click(Pattern("1752605073193.png").targetOffset(47,2))

def SearchLocation(mode):
    # Foco en caja de búsqueda
```

```
do("Open search box", lambda: safe_click("1751993013582.png", 1.0, "Open search box"))
# Limpiar
do("Clear input", lambda: [type("a", Key.CTRL), type(Key.DELETE), wait(0.5)])
# Pegar desde clipboard (ya preparado antes)
do("Paste name", lambda: safe_type("v", "Paste name", Key.CTRL))
# Cerrar sugerencias si procede
if exists(Pattern("1754850754147.png").similar(0.30), Wait_time4):
    #Pattern("1752622220776.png").similar(0.45)
    safe_click(Pattern("1754850754147.png").similar(0.30), Wait_time2, "Search location")

# Desambiguación para Network_diagram
if exists(Pattern("1751972271215.png").similar(0.40), Wait_time) and mode == "Network_diagram":
    click(Pattern("1751972271215.png").similar(0.40).targetOffset(4,54))
    click(Pattern("1751993013582.png").targetOffset(145,-1))
    click(Pattern("1751976333420.png").similar(0.36).targetOffset(-383,-250))

if mode == "Network_diagram" and exists(Pattern("1751977259495.png").similar(0.50),
Wait_time4):
    click(Pattern("1751982526876.png").similar(0.90))
    click(Pattern("1751982236827.png").targetOffset(-59,3))

# Scale adjust in Network Diagram
if mode == "Network_diagram":
    do("Click scale", lambda: safe_click("1751992054822.png", 1.0, "Click scale", target=(-703,490),
similar=0.60))
    do("Set scale", lambda: [type("a", Key.CTRL), type(Key.DELETE), type("1.6")])

def SearchNameWPM():
    wait(0.1)
    if not exists(Pattern("1752490547688.png").targetOffset(-87,2), Wait_time4):
        if exists(Pattern("1752489543538.png").similar(0.90), Wait_time2):
            rightClick(Pattern("1752489543538.png").similar(0.90))
        if exists("1753048423594.png", Wait_time):
            click(Pattern("1753048423594.png").targetOffset(-39,42))
        elif exists("1753090994837.png", Wait_time):
            click(Pattern("1753090994837.png").targetOffset(-38,27))
    if exists(Pattern("1752490547688.png").targetOffset(-87,2), Wait_time4):
        click(Pattern("1752490547688.png").targetOffset(-87,2))
        type("a", Key.CTRL); type(Key.DELETE); wait(0.5); type("v", Key.CTRL)
        waitVanish(Pattern("1753091860079.png").similar(0.40), Wait_time5)
        waitVanish(Pattern("1753088998471.png").similar(0.40), Wait_time5)
        doubleClick(Pattern("1753048618278.png").similar(0.30).targetOffset(-36,-127))

def IsToolbarClosed():
    return (not exists("1751877851392.png", Wait_time)) or (not exists("1751890268170.png",
Wait_time))

# Open apps if not already opened
def GeoViewOpened():
    if not exists(Pattern("1752619159293.png").similar(0.35), Wait_time):
        Open_Toolbar(); Choose_Toolbar("GeoView"); waitVanish("1752064071467.png", Wait_time5)
        Close_Toolbar(); MaximizeScreen()

def NetworkDiagramOpened():
    if not exists(Pattern("1753040718915.png").similar(0.35), Wait_time):
        Open_Toolbar(); Choose_Toolbar("Network_Diagram")
```

```

    Close_Toolbar(); MaximizeScreen()

def WorkPackageManagerOpened():
    if not exists("1753088346194.png", Wait_time):
        Open_Toolbar(); Choose_Toolbar("Work_Package_Manager")
        Close_Toolbar(); MaximizeScreen()

def OMSOpened():
    if not exists("1753129824252.png", Wait_time):
        Open_Toolbar(); Choose_Toolbar("Outstanding_Calls")
        Close_Toolbar(); MaximizeScreen()

# =====
# TESTS SELECTIONS
# =====

def SelectionSingleBatch():
    Single, Batch = 0, 0
    frame = JFrame("Select Single Tests or Batch Tests"); frame.setSize(1000, 500);
    frame.setLayout(GridLayout(0,2))
    cb1 = JCheckBox("Single Tests Session"); cb2 = JCheckBox("Batch Tests Session")
    frame.add(cb1); frame.add(cb2)
    def on_ok(e): frame.dispose()
    frame.add(JButton("OK", actionPerformed=on_ok))
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    frame.setLocationRelativeTo(None); frame.setVisible(True)
    while frame.isVisible(): wait(0.2)
    if cb1.isSelected() or cb2.isSelected(): notify("Test Mode Selected")
    else: notify("No selection. Exiting."); exit()
    if cb1.isSelected(): Single = 1
    if cb2.isSelected(): Batch = 1
    return Single, Batch

def TestSelection():
    flags = [0]*10
    frame = JFrame("Select Tests"); frame.setSize(1000, 500); frame.setLayout(GridLayout(0,2))
    labels = ["Network Diagram Test", "GeoView Test", "Work Package Manager Test", "Trace
Test", "Safety Documents Test",
    "PowerOn OMS Test", "PC Call Supervisor Test", "CallTaker Test", "Trends Test", "Alarms
Test"]
    cbs = [JCheckBox(l) for l in labels]
    for cb in cbs: frame.add(cb)
    def on_ok(e): frame.dispose()
    frame.add(JButton("OK", actionPerformed=on_ok))
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    frame.setLocationRelativeTo(None); frame.setVisible(True)
    while frame.isVisible(): wait(0.2)
    if any(cb.isSelected() for cb in cbs): notify("Running Tests")
    else: notify("No Test selected. Exiting."); exit()
    for i, cb in enumerate(cbs):
        if cb.isSelected(): flags[i] = 1
    return tuple(flags) # Test1..Test10

# =====
# BATCH TEST
# =====

```

```
def BatchTest1(name, retries=0):
    start = time.time()
    result, error, failed_step, screenshot = "Fail", "", "", ""

    for attempt in range(retries + 1):
        try:
            do("Open ND", lambda: NetworkDiagramOpened())
            do("Check Navigation", lambda: CheckNavigationMode())
            do("Copy to clipboard", lambda: copy_to_clipboard(name))
            do("Apply filters", lambda: FilterForSubstation())
            do("Search location ND", lambda: SearchLocation("Network_diagram"))
            # checkpoint: ancla que confirma que el diagrama está cargado
            do("Verify diagram", lambda: require_exists("1751992054822.png", 2.0, "Diagram loaded",
similar=0.40))
            result, error, failed_step, screenshot = "Pass", "", "", ""
            break

        except TestError as te:
            result, error, failed_step = "Fail", str(te), te.step
            screenshot = te.screenshot or snap("auto_fail")
            try:
                CloseWindow()
            except:
                pass
            if attempt < retries:
                wait(0.8)
                continue

        except Exception as e:
            result, error, failed_step = "Fail", str(e), "UNKNOWN"
            screenshot = snap("unknown_error")
            try:
                CloseWindow()
            except:
                pass
            if attempt < retries:
                wait(0.8)
                continue

    duration = round(time.time() - start, 3)
    return result, error, failed_step, screenshot, duration

# =====
# MAIN
# =====

SingleMode, BatchMode = SelectionSingleBatch()
INTERACTIVE = (SingleMode == 1)

# Preconfig común
if not os.path.exists(RESULTS_FOLDER):
    os.makedirs(RESULTS_FOLDER)

# SINGLE MODE
if SingleMode == 1:
```

```

Open_Toolbar(); Close_Toolbar(); Move_Symbol()
(Test1, Test2, Test3, Test4, Test5, Test6, Test7, Test8, Test9, Test10) = TestSelection()

# Network Diagram (single)
if Test1 == 1:
    NetworkDiagramOpened()
    CheckNavigationMode()
    substation_ND = input("Please enter the name of the substation you want to search for:", "",
"Substation Search")
    if not substation_ND:
        notify("No substation name entered. Exiting.")
        exit()
    copy_to_clipboard(substation_ND)
    FilterForSubstation()
    SearchLocation("Network_diagram")
    notify("Substation found in Network Diagram")
    CloseWindow()
    notify("Network Diagram Tests finalized")

# BATCH MODE
if BatchMode == 1:
    # Validar dataset
    if not os.path.exists(SUBSTATION_LIST):
        notify("Substation list not found. Check path."); exit()

    Open_Toolbar(); Close_Toolbar(); Move_Symbol()

    (Test1, Test2, Test3, Test4, Test5, Test6, Test7, Test8, Test9, Test10) = TestSelection()

# --- Batch para Test1 ---
if Test1 == 1:
    timestamp = time.strftime("%Y%m%d_%H%M%S")
    results_path = os.path.join(RESULTS_FOLDER, "Test1_Results_%s.csv" % timestamp)
    # Cargar lista
    substations = []
    try:
        f = open(SUBSTATION_LIST, 'r')
        try:
            reader = csv.reader(f)
            # si hay cabecera
            header_peek = next(reader, None)
            if header_peek:
                # Detecta si la cabecera es texto
                if header_peek and any(s.isalpha() for s in "".join(header_peek)):
                    pass
                else:
                    substations.append(header_peek[0].strip())
            for row in reader:
                if row and row[0].strip():
                    substations.append(row[0].strip())
        finally:
            f.close()
    except Exception as e:
        notify("ERROR reading substation list: " + str(e)); exit()

```

```
# Crear CSV resultados si no existe
new_file = not os.path.exists(results_path)
try:
    f_out = open(results_path, 'ab') if not new_file else open(results_path, 'wb')
    try:
        writer = csv.writer(f_out)
        if new_file:
            writer.writerow(["Substation", "Result", "Error", "Failed
Step", "Screenshot", "Duration", "Timestamp"])
        finally:
            f_out.close()
    except Exception as e:
        notify("ERROR creating results file: " + str(e)); exit()

# Ejecutar lote
for idx, sub_name in enumerate(substations, start=1):
    print("=== [%d/%d] %s ===" % (idx, len(substations), sub_name))
    result, error_msg, failed_step, shot, duration = BatchTest1(sub_name, retries=0)
    timestamp = time.strftime("%Y-%m-%d %H:%M:%S")
    # Append fila
    try:
        f_out = open(results_path, 'ab')
        try:
            writer = csv.writer(f_out)
            writer.writerow([sub_name, result, error_msg, failed_step, shot, duration, timestamp])
        finally:
            f_out.close()
    except Exception as e:
        print("[ERROR] writing results:", str(e))
    notify("Batch Test1 finished. Results saved at: " + results_path)
```