machines

MDPI

Article

A Unified Framework for Automated Testing of Robotic Process Automation Workflows Using Symbolic and Concolic Analysis †

Ciprian Paduraru *,‡, Marina Cernat ‡ and Adelina-Nicoleta Staicu ‡

Computer Science Department, University of Bucharest, 050663 Bucharest, Romania; marina.cernat@unibuc.ro (M.C.); nicoleta.staicu@unibuc.ro (A.-N.S.)

* Correspondence: ciprian.paduraru@unibuc.ro

† This paper is an extended version of our paper published in Paduraru, C.; Cernat, M.; Staicu, A. N. Concolic execution for RPA testing. In Proceedings of the 2023 27th International Conference on Engineering of Complex Computer Systems (ICECCS), Toulouse, France, 14–16 June 2023.

‡ These authors contributed equally to this work.

Abstract: Robotic Process Automation is a technology that replicates human interactions with user interfaces across various applications. However, testing Robotic Process Automation implementations remains challenging due to the dynamic nature of workflows. This paper presents a novel testing framework that first integrates symbolic execution and concolic testing strategies to enhance Robotic Process Automation workflow validation. Building on insights from these methods, we introduce a hybrid approach that optimizes test coverage and efficiency in specific cases. Our open-source implementation demonstrates that automated testing in the Robotic Process Automation domain significantly improves coverage, reduces manual effort, and enhances reliability. Furthermore, the proposed solution supports multiple Robotic Process Automation platforms and aligns with industry best practices for user interface automation testing. Experimental evaluation, conducted in collaboration with industry, validates the effectiveness of our approach.

Keywords: robotic process automation; symbolic execution; concolic testing; workflow testing; automated user interface testing; robotic process automation debugging; software quality assurance

Academic Editor: Philip Moore

Received: 14 April 2025
Revised: 3 June 2025
Accepted: 7 June 2025
Published: 9 June 2025

Citation: Paduraru, C.; Cernat, M.; Staicu, A.-N. A Unified Framework for Automated Testing of Robotic Process Automation Workflows Using Symbolic and Concolic Analysis. *Machines* **2025**, *13*, 504. https://doi.org/10.3390/machines13060504

1. Introduction

Robotic Process Automation (RPA) is a technology that can replicate human interactions at the user interface (UI) level when using various applications. It is one of the newest and fastest growing segments in the software market [1]. Based on the size and value of the company, market share, scope and depth of the solution, the UiPath platform is probably the leading provider of RPA tools [2] and, according to our analysis, offers the most sophisticated support for RPA testing. In addition to UiPath, there are other RPA vendors on the market, such as Automation Anywhere, Blue Prism, Datamatics, Microsoft, Automation Edge, and many more for which the framework described in the following chapters can also be used, but for this article UiPath has been chosen as an example. However, despite its widespread adoption, the methods used to verify and validate RPA workflows have not evolved at the same pace. In practice, testing remains largely manual or relies on fragile black-box techniques, resulting in limited test coverage, high maintenance effort, and a higher risk of undetected errors in production systems. This research addresses the lack of effective, automated white-box testing techniques tailored specifically for RPA workflows. Existing approaches typically do not analyze the internal control-flow structure

of workflows, nor do they leverage formal methods to improve test generation or reliability. Against this backdrop, our work is guided by the following research questions:

- How can symbolic and concolic execution techniques be adapted and applied to RPA workflows for white-box testing?
- Can a hybrid strategy improve test coverage and fault detection compared to existing black-box or ad hoc methods?
- What are the practical implications and benefits of applying such a framework in real-world RPA development environments?

To investigate these questions, we propose the following hypothesis:

Integrating symbolic and concolic execution techniques into an RPA testing framework can lead to improved test coverage and fault detection, while reducing manual testing effort and maintenance costs.

To test this hypothesis, we developed an open-source testing framework combining symbolic and concolic analysis for RPA workflows. The framework was evaluated on real-world use cases provided by industry partners, allowing us to assess both its effectiveness and practicality in real development settings.

The main contribution of this article can be summarized as follows:

- We gather various methods of testing RPA systems and we implement them in a single open-source package at https://github.com/unibuc-cs/rpa-testing (accessed on 6 June 2025). While the evaluation and results presented are based on RPA processes developed using the UiPath Studio tool, the proposed architecture—designed with a clear separation of concerns—allows for the integration of other similar tools with minimal effort.
- A novel greedy strategy, referred to as All states ones, is introduced by this paper in Section 4.3.
- We validate our ideas and prototypes in collaboration with industry partners. In particular, in our work with our industry partners at UiPath, and based on experiments, we identified a suite of best practices to be used for testing RPA scenarios. These practices incorporate adapted methodologies from the software testing literature, such as mocking, human-in-the-loop testing, and white-box/black-box fuzzing. The evaluation is supported by prototype implementations across a variety of industrial RPA use cases.

The proposed framework is based on a few assumptions that support its effectiveness in structured automation environments. It assumes the availability of control-flow graphs and basic annotations, which are commonly present in modern RPA development platforms and help guide the path exploration process. The workflows considered in this study come from real-world domains such as banking and healthcare, but are of moderate complexity; while they reflect practical scenarios, larger or more dynamic systems may present additional challenges. The selection between the execution strategies is currently performed manually, which allows flexibility depending on coverage goals and time constraints.

The remainder of the article is organized as follows. The next section reviews prior research on RPA, as well as broader work on software testing techniques. Section 3 provides an overview of RPA concepts, tools, and the motivation for automating the testing of UI application using RPA technologies, including illustrative examples. Section 4 details the testing of RPA workflows using the All states once coverage strategy and further elaborates on workflow testing through symbolic and concolic execution methodologies, respectively. The evaluation of these methods on real-world applications, along with a discussion of best practices, is extended in Sections 5 and 6. Finally, the last section, Section 7, summarizes the main findings of the research, discusses its limitations, and suggests possible directions

for future investigations, including enhancements to the proposed methodologies and their applications in different industrial contexts.

2. Related Work

After studying the literature on RPA and testing, we found that there is no prior work on white-box testing for RPA workflows. Ref. [3] describes the current level of software robot testing and provides ideas for automation using model-based testing. However, they do not provide further evaluation or implementation proposals. Ref. [4] created a testing environment by observing user interactions with application UI interfaces and documenting them with logs and photographs. This method falls under "black-box testing", as it does not analyze an RPA workflow. Although fascinating, it has shortcomings when compared to our proposed methods. (a) It is more challenging to bring user information into the test field, and (b) in contrast to white-box testing of a given graph, recording actions solely using computer vision techniques might severely restrict the coverage of the test state space. Second, graphical user interfaces were tested using RPA in the papers by [3,5]. Furthermore, ref. [6] employed RPA for regression testing and automation for more typical information system components, such as desktop, mobile, or web applications, or even for large-scale software's API functionality testing. In the studies [5,7], graphical user interfaces were tested using RPA. Furthermore, ref. [6] employed RPA for regression testing and automation for more common information system components, such as desktop, mobile, or web applications, or even for API functionality testing in large-scale software.

Our testing methodology aligns with contemporary advancements in white-box testing, particularly emphasizing symbolic execution techniques. Recent studies have highlighted the integration of large language models (LLMs) in white-box testing to enhance test case generation and optimize testing processes. For instance, the development of WhiteFox demonstrates the application of LLMs in compiler fuzzing, leading to improved detection of compiler defects [8]. In the realm of symbolic execution, tools like FuSeBMC have been developed to combine fuzzing and symbolic execution, applying bounded model checking to identify security vulnerabilities in C programs [9]. Additionally, approaches such as TracerX [10] enhance dynamic symbolic execution by incorporating interpolation methods to manage path explosion and improve verification efficiency. A recent survey by [11] provides an overview of new research trends in symbolic execution, with particular emphasis on applications to test generation. These advancements have inspired our approach to repurpose cutting-edge techniques and contribute to the field through research. We assess and offer open-source versions of the suggested modifications to make them suitable for the Robotic Process Automation (RPA) testing domain. By examining and adapting earlier research focused on source code, Low Level Virtual Machine (LLVM), or binary testing, we aim to develop effective testing methods for RPA processes. In particular, the concept of symbolic execution has been applied to various levels of program representation, including source code and LLVM. For instance, tools like Concolic Replay and Testing (CRETE) [12] operate at the LLVM representation level, providing insights that have influenced our annotation strategies, which are discussed in subsequent sections. By leveraging these contemporary tools and methodologies, we aim to enhance the robustness and reliability of RPA testing processes.

In our study, ref. [13] delves into the potential of Robotic Process Automation (RPA) within the gaming industry, illustrating how automation technologies can address operational inefficiencies and enhance productivity. The authors highlight RPA's ability to streamline repetitive and error-prone processes, such as software testing, quality assurance, and data validation, which are integral to game development and maintenance. Further-

more, the study emphasizes RPA's role in augmenting user satisfaction by ensuring faster issue resolution and improved in-game experience.

Key areas of focus include the scalability of automation solutions in dynamic gaming environments and their contribution to cost reduction and operational resilience. Ref. [13] also outlines practical implementation frameworks and strategies for integrating RPA into existing workflows, providing insights for both industry professionals and academic researchers. By leveraging RPA, the gaming sector stands to benefit from optimized processes, reduced time-to-market for games, and a more adaptive approach to emerging challenges.

This work serves as a foundational reference for exploring automation in gaming and underscores the strategic importance of RPA in an increasingly competitive and technology-driven market. While prior work has explored testing in the context of RPA, we observed that the vast majority of these efforts focus on black-box methods or use RPA tools to test external systems. To our knowledge, no existing research has addressed white-box testing approaches that are specifically tailored to RPA workflows. In particular, the internal control-flow structure of workflows remains largely unexplored in test generation strategies. This lack of attention to white-box techniques in RPA represents a clear research gap. Our work aims to fill this gap by adapting symbolic and concolic execution methods to the specifics of RPA, offering a new direction for improving test coverage and reliability in this domain.

3. An Introduction to RPA and Test Automation

UiPath Studio (https://docs.uipath.com/studio/standalone/latest, accessed on 6 June 2025) is the environment in which the robots are visually designed and provides three primary workflow types that can be used to visualize the order in which a robot performs tasks: Sequence, Flowchart and State Machine. Examples of these can be found in this document in Figures 6–8. In the sequence, the activities are presented in a linear fashion. The flowcharts, on the other hand, provide greater flexibility with multiple decision points and arrows that connect any two actions. They are comparable to BPMN (Business Process Model and Notation) models, a graphical standard for modeling business processes that visually represents workflows using symbols such as events, activities, and gateways, as well as Unified Modeling Language (UML) activity diagrams. The state machines offer even more expressive power than flowcharts, as they support conditional transitions and internal states for nodes, making them similar to traditional UML state machines.

Data can also be incorporated into the aforementioned forms as local variables or workflow arguments. Additionally, each workflow can be hierarchically embedded within another workflow, allowing for greater modularity and reusability. For example, a workflow can contain a series of flowcharts with local state machines in certain nodes. When creating the workflow, existing basic activities, libraries implemented in source code (e.g., `C#` code fragments) or other workflows are inserted using drag and drop. Basic functions include reading and writing data to and from a variety of formats (PDFs, Excel, Word, common databases, desktop or web-based applications), working with all kinds of variables (since UiPath Studio is based on the .NET framework), creating reports, managing mouse clicks and keystrokes, and even using optical character recognition (OCR) to convert handwritten or printed text, as well as text contained in images, into machine-readable text. To create workflows, the user has the option to create them manually in the UiPath Studio interface, use the recording function or choose a hybrid of both. UiPath recording is a feature that captures user interactions with applications, such as mouse clicks, keyboard inputs, and UI navigation, to create automation workflows. It simplifies the process of building automation by generating sequences of activities that mimic human actions. UiPath offers

different recording types, including Basic, Desktop, Web, Image, and Native Citrix, each optimized for different environments. Once recorded, the automation can be modified, enhanced with logic, and executed by robots to streamline repetitive tasks efficiently.

We chose UiPath as the RPA provider because it offers a robust and flexible environment suited for implementing symbolic and concolic execution techniques tailored to RPA workflows. The selection criteria included the platform's support for workflow customization, accessibility of internal workflow structures, compatibility with our testing framework, and its widespread use in industry, which ensures practical relevance. While other RPA platforms could potentially be adapted to our approach, UiPath provided the best balance of technical features and ease of integration for this study.

3.1. Improving UI Test Automation with RPA

Automated tests for Robotic Process Automation (RPA) can significantly improve user interface (UI) testing. UI testing is often considered difficult due to various factors. Modern applications are often constantly updated to incorporate new features and functionalities, making comprehensive UI testing a challenge [14]. Furthermore, the increasing complexity of applications with features such as complex flowcharts, maps, embedded frames and diagrams further complicates UI testing [15]. Moreover, the dynamic nature of UI elements, which change frequently, leads to difficulties in maintaining UI test scripts [16]. When multiple or remote applications are involved in UI testing, additional challenges arise, such as limited control over graphical user interface (GUI) elements compared to testing a single web application with tools such as Selenium [17]. RPA can solve some of these problems by working effectively at the user interface level in heterogeneous environments. It often works without scripts, making it more accessible, and uses advanced technologies such as computer vision to improve stability and performance [18].

RPA has both opportunities and challenges, with many initiatives failing despite its potential benefits. Specifically, RPA's development life cycle is still in its early stages since it is a new paradigm. A key component is quality assurance, a laborious process that is essential to a project's success. The most popular method for ensuring that the deployed robots are operating as intended and with few faults is testing. Nevertheless, the majority of RPA testing is now carried out manually, and there are not many tools available to help with it. Based on the following examples, a discussion can be opened on the topic of RPA testing and, more precisely, how the state of the practice can be improved with state-of-the-art research in testing. In this way the study of automation in RPA testing can provide new research topics.

Based on the following examples, a discussion can be initiated on the topic of RPA testing, specifically focusing on how the current state of the practice can be enhanced by state-of-the-art research in testing.

3.2. Motivating Examples and Current Practice

3.2.1. Testing a Calculator Using RPA

Among the various available options, test automation can now also be achieved through RPA. A small example of this implementation can be found in the article "Improving UI Test Automation using Robotic Process Automation" [7], where two calculator apps were tested—one installed locally and the other accessible online. For this case, a robot was developed to test both calculators in parallel using test data stored in an Excel spreadsheet. The robot automatically interacts with the interfaces of both calculators, implementing a data-driven testing approach, extracts the results, and compiles them into a file for test reporting. Unlike traditional UI web testing frameworks, such as Selenium, which are

limited to working with browser applications, the robot is capable of switching between all three applications (the web calculator, the desktop calculator, and the Excel spreadsheet).

3.2.2. Testing the Back-Office of a Banking Application Using RPA

Another example of end-to-end testing using an unattended RPA robot can be found in the back-office operations of a banking application. Initially, the goal is to verify if the application functions correctly across multiple web browsers. The robot should be able to log in successfully regardless of the browser being used. Once the robot evaluates the login functionality, it proceeds to test the internal money transfer feature. The robot verifies whether the transferred amount reaches the intended account. If the transfer is successful, the test passes, and the outcome is saved. After the payment is completed, both the bank clerk and the robot should be able to view and store the payment confirmation in PDF format. To test this feature, the robot navigates to the menu, locates the payment confirmations, and saves the most recent file. It then opens the PDF file and checks the relevant fields to ensure it includes the transfer amount and the recipient account. If these fields are accurate and present, the test is deemed successful.

3.2.3. An Application for a Bank Loan

Consider a simple procedure (adapted from UiPath's documentation) that determines the type of bank loan based on its amount and duration.

Figure 1 shows the flowchart of the Create_Loan process, which contains three decisions (represented by a '?' and a condition). The first two relate to the loan amount and change the flow direction in a branch based on the loan category (low, medium or high). The last decision is about how long the loan can run for. The loan is considered short-term if it has a term of less than, say, five years; otherwise, it is considered long-term.

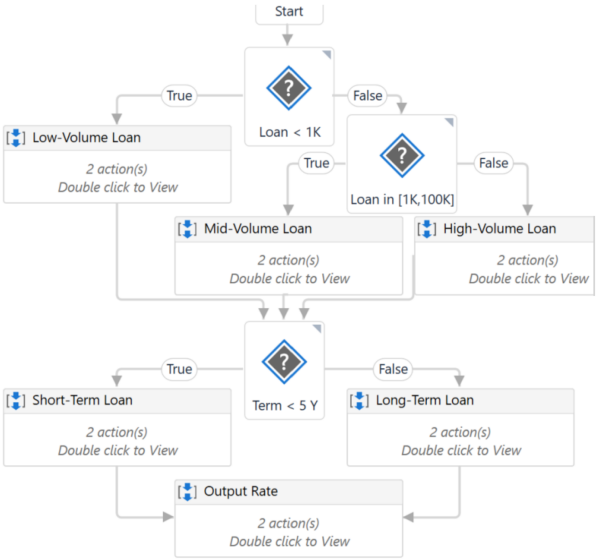

Figure 1. The main workflow of the BankLoan application.

Another, more complex example involves rule-based, high-volume processes in the retail industry. Rule-based, high-volume procedures are commonly automated in the retail sector, with refunding returned orders being a typical B2C (business-to-consumer) operation. This process often requires significant human resources, particularly during peak sales periods. By automating refund tasks with robots, businesses can save time, improve compliance, and optimize resource allocation. The robot's input is a web-based report containing details such as sales order numbers, customer account data, return dates,

and more. After logging into a customer relationship management (CRM) system, the robot retrieves each sales order number from the input report. It then identifies which items are eligible for a refund and calculates the refund amount for each request. If the customer has a voucher, the robot applies any relevant discounts. Once the total refund amount is determined, the robot accesses a second application, a payment platform, to transfer the full refund amount to the customer. The robot generates an output Excel file containing the sales order numbers, refunded amounts, and the status of each activity.

Current RPA testing techniques either rely on black-box UI observation or unstructured script-based testing. There is no existing symbolic or concolic execution engine tailored for workflow-based RPA environments with visual activity models. Our work fills this gap by adapting symbolic [19] and concolic analysis techniques to the structure of RPA workflows and implementing a scalable, open-source testing engine based on SMT solvers such as Z3 [20]. Symbolic execution is a program analysis technique that explores program behavior by determining the input values required to execute different code paths. While it theoretically offers high code coverage, practical adoption is often limited by the path explosion problem, where the number of feasible execution paths grows exponentially with program complexity.

Two principal variants of symbolic execution are employed in software testing: online symbolic execution and offline symbolic execution, also known as concolic execution.

In concolic execution, the program is executed with concrete inputs while symbolically tracking path constraints encountered at conditional branches. These constraints, representing logical conditions on symbolic inputs, are recorded as execution traces. At the end of each run, these traces are analyzed offline to generate new input values by solving the collected path constraints, enabling systematic exploration of alternative execution paths. This decoupling of path exploration and execution makes concolic execution more scalable and better suited for large and complex software systems compared to pure online symbolic execution.

These challenges underscore the need for more systematic, scalable, and automated testing methodologies in the RPA domain. While current RPA testing is largely manual and tool-limited, the dynamic and heterogeneous nature of UI workflows demands techniques that can adapt and reason about workflows structurally. This motivates our use of white-box testing methods, particularly symbolic execution and concolic analysis, to explore the workflow's internal logic and generate test cases with high coverage and minimal manual intervention. In the remainder of this paper, we formalize this approach and validate it through open-source implementation and empirical evaluation.

4. Methods

In this section, the first two subsections provide an overview of the known concolic and symbolic testing methods applied to the RPA domain, summarizing the current state of the art. We then present the technical details and insights gained during the implementation of these methods in the proposed open-source package, which constitutes the primary contribution of this paper. In Section 4.3, we introduce a novel technique, a hybrid of the previous two methods.

4.1. RPA Testing Using Symbolic Execution

The methods described below assume that the RPA workflow is represented as a directed graph G, where each action is modeled as a node. This graph may include cycles, with the starting action (i.e., node) designated as Initial(G). The decision factor for generating and executing test cases is primarily based on branch points, which represent activities that vary depending on specific conditions.

Algorithm 1 illustrates the main loop of the implementation pseudocode within our framework. The process begins at the initial activity node of the graph G (Line 2), inserts it with a single-node into a worklist W, extracts a promising path at each step of the loop (with customizable strategies for evaluating path priorities), and moves forward to Line 7. The worklist W is a priority queue sorted by priority of the element (path). The default priority calculation is determined based on the path's depth in the exploration tree of G. The algorithm prioritizes paths at the top of the exploration tree for quicker traversal, rather than focusing on long paths with frequent activity nodes. Our framework allows users to alter default behavior with function hooks according to specific testing goals.

Algorithm 1 The main loop of Symbolic Execution using DFS strategy.

1: `// Initialize the starting exploration path with the entry node of the workflow graph`
2: $start\_path \leftarrow \text{SMTPath}([G.entry\_node], \text{priority} = \text{MAX\_PRIORITY})$
3: `// Add the initial path to the worklist`
4: $W.\text{add}(start\_path)$
5: **while** $W \neq \varnothing$ **do**
6: $\quad currPath \leftarrow W.\text{extract}()$
7: $\quad \text{EXECUTEPATH}(currPath, W)$
8: **end while**

New entries in the work queue W are prioritized based on their depth index in the exploration tree, where the branching constraint is reversed. When sorting the priority queue W by priority, the default behavior favors modifications to paths that diverge closer to the root of previously explored paths. This approach is also employed in [21] and promotes broader early exploration. However, the user can override this behavior by providing a custom hook function if different prioritization is desired.

Algorithm 2 presents the code responsible for continuing the execution of a given path. The SMTPath class abstracts the execution path in this symbolic form. Line 3 adds an initial set of restrictions on variables in the DataStore (D), such as those propagated from annotations, to the SMT solver's state. The implementation parses the current node to make the continuation decision (Line 4).

When a branch node occurs (Lines 8–31), the current path is cloned and split into two; one takes the True branch and the other takes the False branch. To create a future worklist, we prioritize possible paths and assign the next node based on the branch under consideration. Line 27 shows a redundant DataStore (D) object for one path. This guarantees that each uses separate data context values. This enables concurrent execution of worklist branches. The implementation follows a depth-first search (DFS) approach, with the currently active path taking precedence over the other. The latter path is added to the worklist and may be run later (Line 29). Some implementations use a breadth-first-search (BFS) method, where both branches are added to the worklist based on priority, and a new path is explored at each step. Figure 2 illustrates the difference between the two.

Algorithm 2 Continuation of execution for a given path.

```
1: procedure EXECUTEPATH(path : SMTPath, W : Worklist)
2:     currNode ← path.nextNodeToExecute ▷ Each path stores the next node to execute (initial is entry node)
3:     path.conditions_smt ← DataStoreTemplate.getVariablesAssertions() ▷ Add constraints from DataStore annotations
4:     while currNode ≠ None do
5:         if currNode.type ≠ BRANCH then
6:             WorkflowExecutor.Execute(currNode)
7:             currNode ← currNode.next()
8:         else
9:             if not HasSymbolicVariables(currNode.condition) then
10:                Result ← WorkflowExecutor.Execute(currNode)
11:                currNode.advance(Result)
12:                continue
13:            end if
14:            newPath_onTrue ← SMTPath(
           nodes = currPath.nodes + currNode.nextNode(True),
           conditions_smt = currPath.conditions_smt + currNode.condition
15:            )
16:            newPath_onFalse ← SMTPath(
           nodes = currPath.nodes + currNode.nextNode(False),
           conditions_smt = currPath.conditions_smt + Not(currNode.condition)
17:            )
18:            solvableNewPaths ← []
19:            if SMTSolver(newPath_onTrue) has solution then
20:                solvableNewPaths.add(newPath_onTrue)
21:            end if
22:            if SMTSolver(newPath_onFalse) has solution then
23:                solvableNewPaths.add(newPath_onFalse)
24:            end if
25:            for all newPath in solvableNewPaths do
26:                newPath.scorePath()
27:                newPath.data_store ← currPath.data_store.clone()
28:                newPath.nextNodeToExecute ← currNode.nextNode(True if newPath = newPath_onTrue else False)
29:                W.add(newPath_onFalse)
30:            end for
31:            break
32:        end if
33:        StreamOut(currPath)
34:    end while
35: end procedure
```

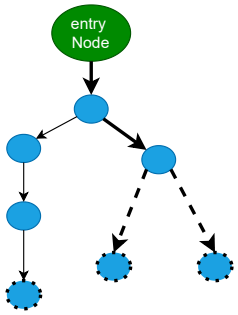

Figure 2. The distinction between the DFS and BFS execution types is depicted in the figure. Broken circles are used to symbolize frontier nodes, and visited nodes are those that remain. The route that is emphasized is the one that is currently being followed. The next iteration of BFS-style execution chooses the top continuation node (by priority), which may even be the unexplored node on the left side of the workflow graph. Both new boundary nodes are added to the priority queue worklist. One of the two new nodes is added to the work queue in the DFS manner, and additional execution proceeds on it.

4.2. Concolic Execution for RPA Testing

Our review of the literature revealed a notable gap in testing methodologies tailored to RPA workflows of varying complexity. To the best of our knowledge, no prior work has thoroughly addressed this issue. While the article "Towards Automated Testing of RPA Implementations" and the preceding section offer initial suggestions for automating testing through model-based techniques, they lack further analysis, reusable implementations, and technical depth. Our approach builds on whitebox testing principles. We explore, evaluate, and provide open-source implementations of the proposed adaptations to ensure their applicability to the RPA domain, leveraging state-of-the-art techniques. Specifically, we revisit and tailor prior research originally designed for source code, LLVM, or binary testing, adapting it to meet the unique needs of RPA workflows.

4.2.1. Overview of the Testing Pipeline

The RPA process specification undergoes a series of transformations before being handed off to specialized fuzzing techniques tailored to our use cases. The entire workflow is illustrated in Figure 3. Initially, the user builds an RPA workflow (W0) and an annotation set (S) with a variety of parameters. These inputs are then bundled and passed to a process called WorkflowParser, which parses W0 and S to produce a format that is easy to grasp for the subsequent fuzzing operations carried out by the WorkflowExecuter process. At the end of this process, a test suite is returned, which the tested robot can subsequently run. It is important to differentiate that the WorkflowExecuter simulates the state space environment to efficiently generate test cases, while the RPA Robot agent represents the actual machine or process in production. It is also feasible to generate test cases in real time on the robot side, but this would often involve a large latency and is therefore less efficient.

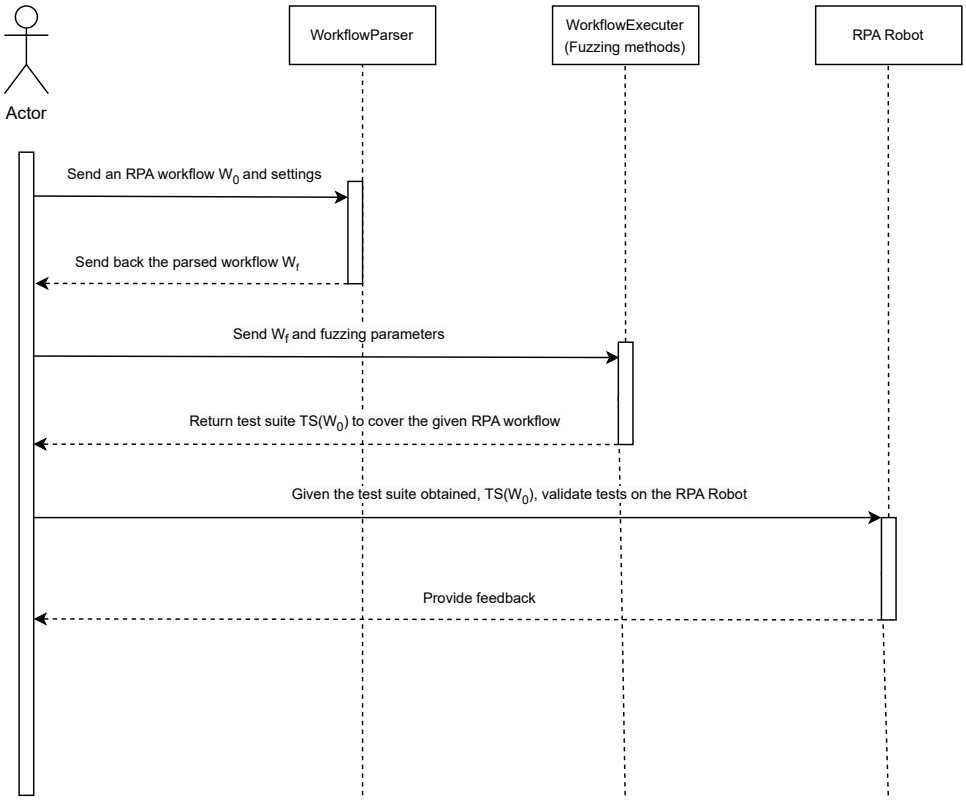

Figure 3. The overview of the full testing process, starting from the RPA workflow W0 given by the user to obtain a set of test cases and validate them on the robot.

Figure 3 provides a comprehensive overview of the full testing process, illustrating the roles and interactions of the various components described in this section.

4.2.2. Parsing Operation

The WorkflowParser procedure takes an RPA workflow W0 (Figure 2) that is generated in a text or visual application and converts it into a ContextFlowGraph before exporting it physically as a JavaScript Object Notation (JSON) file (Wf). The exported format represents each activity in the workflow as a hypernode in the graph G. If there is a decision branch in the activity, then the related nodes in G are linked appropriately. For example, if the activity involves querying databases and setting variables based on the findings, the hypernode is made up of multiple interconnected nodes that build an abstract syntax tree (AST) with low-level operations. Figure 4 illustrates the hypernode formed by an activity with a source code, as shown in Algorithm 1. Our environment supports several types of AST nodes by default, including logical and mathematical expressions, function calls, array and data table access, and variable reading and writing utilizing a DataStore, as described in Figure 5. Our collaboration with industry partners highlighted the importance of allowing users to contribute their own nodes to the project, as different functionalities may be required depending on the workflows being tested. Our research indicates that annotation support is crucial for high-quality testing. Annotation involves adding additional information to activity nodes, such as hints. Annotations can specify limits and patterns, saving testing resources from unnecessary pathways. The human-in-the-loop approach allows users to offer valuable information for fuzzing methods. This reduces computation and creates test cases that are understandable to humans. To ensure visibility for all stakeholders, we recommend setting annotations directly in the RPA workflow editor (see Algorithm 3).

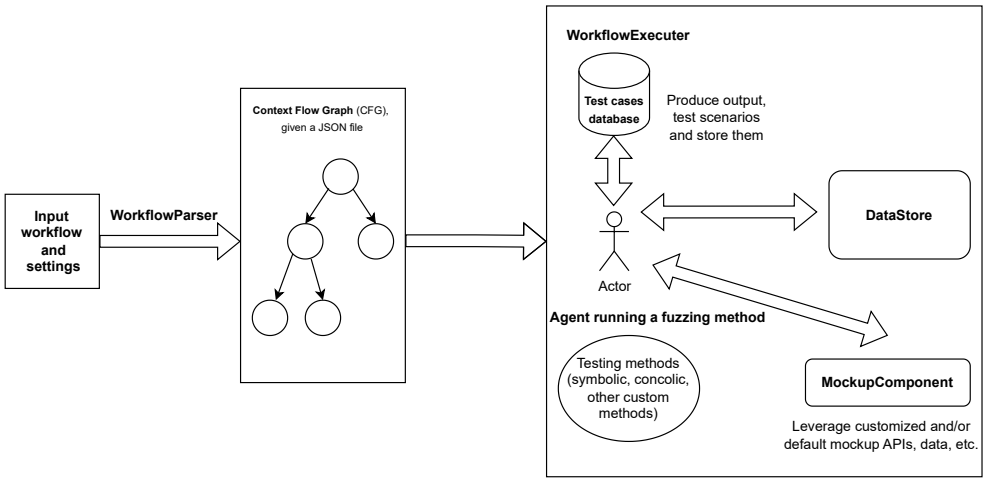

Figure 4. The detailed overview of the steps and components involved in the testing process.

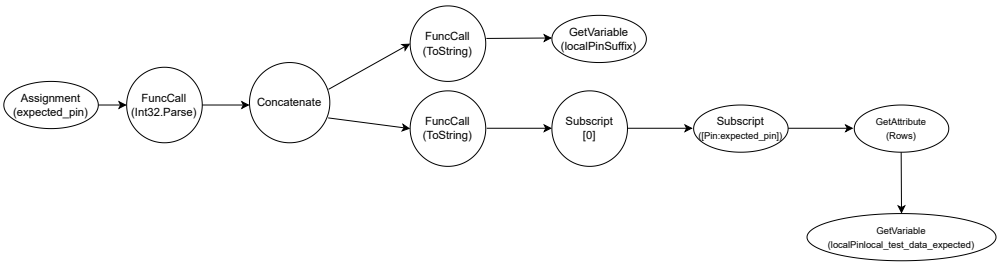

Figure 5. The AST generated internally by the WorkflowParser after evaluating the source code presented in Algoritm 1.

Algorithm 3 Example of an expression contained in an activity node.

```
// Input:  local_test_data_expected, localPinSuffix
// Output:  expected_pin
expected_pin = Int32.Parse(
   local_test_data_expected.Rows[0]["Pin:expected_pin"].ToString()
   + localPinSuffix.ToString())
```

4.2.3. Annotations

In the proposed testing framework, annotations are used to define boundaries and patterns that help the engine avoid executing irrelevant or invalid paths. Annotation refers to the insertion of additional data—such as hints or metadata—into activity nodes. For instance, by applying a human-in-the-loop approach, users can contribute valuable domain knowledge to guide fuzzing techniques. This collaboration not only reduces computational overhead but also leads to more human-readable and meaningful test scenarios. We found that making annotations directly visible in the RPA workflow editor is a practical design choice, helping both developers and testers understand how test data is controlled and interpreted. Currently, our system supports two main categories of annotations:

- Variable-level annotations are defined as additional information associated with variables, such as min/max ranges, pattern restrictions, or user-specified input values. This metadata is used to reduce the search space during test data generation and ensure realistic distributions. It also allows the user to insert CSV mock files to simulate input values for specific variables. For example, the user can set a variable named

actual_pin_values to accept only numbers between 0 and 9999 and also declare it as a user input. A mock data file can then be used to populate the value of this variable during test execution. Algorithm 4 provides an example.

- Source-code-level annotations are used to insert small code snippets that perform additional operations before or after a node is executed. This is mainly used to define assertions or apply mock behavior. Algorithm 5 shows a simple expression annotation that is evaluated as an assertion to verify whether the value of the pin at a certain index equals the expected value. These expressions are written using a domain-specific expression language and are evaluated during test execution.

Algorithm 4 Example: Various support for annotating the variables in the activity.

```
@Variables
{
  "local_number_retries":{"min": 0, "max": 3},
  "local_test_data_expected":{"path": "pin_mocked_data.csv"},
  "local_test_data_actual":{"path": "pin_real_data.csv"},
  "actual_pin_values":{
    "bounds": 10,
    "min": 0,
    "max": 9999,
    "userInput": "True"
  }
}
```

Algorithm 5 Example of additional code sanity check for a PIN number added as annotation to an activity in the workflow.

```
@Expression
{
  "expression":
  "actual_pin_values.GetElementAt(local_number_retries) ==
expected_pin"
}
```

Annotations are processed by the testing component, which takes as input the exported Control Flow Graph (CFG) and annotation metadata in JSON format, both generated by the WorkflowParser component (Figure 3).

A central DataStore (D) holds both concrete values and symbolic representations of variables influenced by user input. In the literature, this influence is referred to as tainting—describing how external inputs (like user actions or network data) can impact variables. The initial set of tainted variables, SymbolicVars (D), is either user-supplied through annotations or automatically detected as external input. As the workflow executes, other related variables become symbolic as well. The DataStore contains a structure Values (D), which maintains the current concrete values for all variables—symbolic or not—throughout evaluation. then executes the CFG node by node, following branches and links. At each step, the DataStore updates both the concrete and symbolic/tainted values to reflect current context and branching conditions. When a node is executed, its functionality—defined by the workflow—is evaluated. These evaluations may have visible side effects in the DataStore. In the abstract syntax tree (AST) of a node (see Figure 4), a

leaf might represent a function like GetVariable, which fetches data from the DataStore. The expression is then propagated to the root of the tree and evaluated. The evaluation of node-defined functionality can cause apparent side effects in the DataStore. Understanding the function and significance of real-time node evaluation is critical. In the matching node AST (Figure 4), a leaf has a GetVariable function call that fetches data from the *DataStore*. The expression is then moved to the top of the tree.

$$\begin{aligned} expected\_pin &= Eval(node.expression) \\ Values(D) &= Values(D) \cup expected\_pin \end{aligned} \tag{1}$$

The WorkflowExecutor is capable of executing the graph with different strategies around decision points or branches.

4.2.4. Mocking

Mocking refers to the practice of replacing real data sources, library calls, or system components with controlled substitutes. This can help reduce test complexity, avoid reliance on external systems, and focus testing on specific components or logic paths. The proposed framework supports three types of mocking:

- Data sources mocking is useful when the workflow relies on remote databases. For example, a distributed SQL (Structured English Query Language) server used in production can be replaced with a local CSV file for testing. The mock file should have the same schema and distribution as the real data. These mock files can be periodically updated or linked to the real database via automated scripts. The framework uses an extensible DataTable component that can be accessed locally or remotely through a RESTful (Representational State Transfer) API (Application Programming Interface). This component can be customized by users to fit their specific requirements.
- API/Library mocking is useful when workflows call external APIs or libraries. Consider the example shown in Algorithm 6, which illustrates an example of such a call. Our framework supports this by offering an extendable table of classes and functions, like doServerRestAPI.getClientSalary, ToString, and getServerTimeOfDay. where an API is used to fetch the salary of a client, convert it to a string, and append a timestamp. This type of logic is common in real-world workflows but may introduce dependencies that are not desirable during testing. In this case, the user can register a mock implementation of the API or library functions, which will be called instead of the real ones. For example, the getClientSalary or getServerTimeOfDay functions can be overridden to return fixed values.
- Data types mocking is useful when working with complex or heavy types, such as System.Threading.Channels in C#. These types can be replaced with mock classes implemented in Python or C#, depending on the execution engine. This allows the user to simulate the behavior of certain components without introducing execution overhead or side effects. This mocking technique is aligned with modern software engineering practices such as dependency injection and inversion of control.

Algorithm 6 Example of multiple library calls within a node in workflow activity.

```
1: // Expression inside activity node
2: out = doServerRestAPI.getClientSalary(loggedID).ToString()
3:         + getServerTimeOfDay().ToString()
```

4.2.5. Generation of Test Cases

Based on evaluations and benchmarking with industry partners, we found that partitioning the generated test cases into smaller units at runtime represents a best practice for constructing extensive test corpus and efficiently leveraging the computational capacity of deployed RPA robots. The granularity of these partitions can be user-defined. This is crucial for continuous integration and regression testing, as the test framework and RPA robots may function in a producer–consumer model. In this setup, the framework acts as the producer by generating test case scenarios, while the RPA robots act as consumers, utilizing them simultaneously.

4.2.6. The Concolic Execution Engine

Our evaluations indicate that symbolic execution is more effective for small-scale RPA tasks. In contrast, concolic execution is better suited for achieving state-activity coverage in modern agile development environments, especially when graph G can be explored through loops or when data-dependent synchronization is required. Our framework incorporates the SAGE (System for Algebra and Geometry Experimentation) method's basic concepts.

To adapt concolic execution to the RPA domain, the proposed test is run through fuzzing with an RPA software robot, identifying branch points along the workflow path. At the end of the execution, the inputs are changed to a different set, resulting in different paths the next time. The software robots are operated remotely by another robot or a server procedure. Algorithm 7 shows the algorithm's main loop. The procedure uses a worklist (line 2) with inputs that span the workflow's variable space. This list prioritizes items based on their potential for increasing coverage, uncovering hidden states, and delving deeper into the state space tree. We allow users to specify the priority definition as an overridable function for maximum flexibility during testing. Common examples in the literature include increasing code coverage, identifying difficult scenarios, and addressing error-prone sections of source code. This inversion of the control technique, combined with annotation support, integrates human expertise into the testing process. The priority queue might begin with an initial set of input data, either randomly generated or from a test corpus. To optimize efficiency, it is crucial to add an initial set of input data to the worklist W during concurrent execution. This allows for early discovery of critical parts of the activity graph. The method would take longer to obtain the necessary coverage rate if it were not optimized. Our version of the architecture incorporates human knowledge into the production of initial input seeds, a feature known as "human-in-the-loop". To start testing, a user-supplied CSV file with inputs for the workflow's variables must be present. Rows specify test cases, and columns indicate the ordered set of variables in the workflow being tested. Missing data is acceptable, and only available data is used. Users can prioritize tests (last column in each row) and set them as key values in the worklist.

Algorithm 7 Main loop of a server or RPA robot for collecting concurrent execution tests.

```
1: // Initialize the starting set of input seeds
2: W ← {any initial method to fill starting set}
3: TestCorpus ← {W}
4: while W ≠ ∅ do
5:     R ← findRobotToExecuteTest()
6:     if R ≠ null then
7:         T_0 ← W.extract()
8:         R.newPath ← SMTPath(T_0)
9:         R.ExecutePath(R.newPath)
10:        NewInputs ← R.GenerateNewInputs(R.newPath)
11:        TestCorpus ← TestCorpus ∪ NewInputs
12:        W ← W ∪ NewInputs
13:    end if
14: end while
15: return TestCorpus
```

Line 5 of the main test coordination loop selects an RPA agent suitable for performing the test. In our usual implementation, this is the first available free agent. The function blocks until one of the robots becomes available. The robot executes the most promising input seed from the worklist (line 9). At this point, the path consists of a series of activities that take place when the input is executed. The path includes information on each branch point, including the SMT condition it operates under. In line 10, this information is used to modify the initial input text, allowing for different execution paths. The resulting TestCorpus includes both the original and newly discovered tests.

Algorithm 8 provides pseudocode for performing a path based on an input. In the workflow graph G, execution starts at the initial node. If the current node is not a branch decision (line 6), the execution will proceed to the next node. The executor evaluates the condition within the node (line 7) and returns True or False accordingly. If the branch node's condition does not rely on user-controlled variables, the execution will proceed with the condition evaluation result, as shown in line 11. If not, both the SMT assertion and its inverse are recorded in the route data structure for future processing (line 16). Finally, execution proceeds. The pseudocode summarizes all branch point assertions that potentially change pathways when the SMT solver changes the malformed input material associated with the branch conditions. It is worth noting that only the branch decision constraints can be changed to affect the execution path. The annotations set restrictions that are read-only (line 6). When a constraint is set due to a branching choice (Lines 8–14), the SMT solver (S) can potentially generate additional pathways. First, S's current stack context is stored (Line 9). Next, the inverse constraint is applied to S. If the modified constraints in S are now satisfiable (Line 12), a new input may lead to a different path of activities. Line 12 updates the set of inputs. To adjust a single branch condition with each new input, restore the previous SMT state and add the path evaluation decisions to the SMT solver. The execution behavior remains unchanged, save for the modification of the branch node for the new path.

Algorithm 8 Executing an input path in concolic mode and creating an `SMTPath` object.

```
1:  procedure EXECUTEPATH(path: SMTPath)
2:      currNode ← path.entryNode
3:      indexBranchAlongPath ← 0
4:      path.conditions_smt ← DataStoreTemplate.getVariablesAssertions()
5:      while currNode ≠ None do
6:          if currNode.type ≠ BRANCH then
7:              WorkflowExecutor.Execute(currNode)
8:              currNode ← currNode.next()
9:          else
10:             Result ← WorkflowExecutor.Execute(currNode)
11:             if ¬HasSymbolicVariables(currNode.condition) then
12:                 currNode.advance(Result)
13:                 continue
14:             end if
15:             if Result = True then
16:                 Taken_assert ← currNode.condition
17:                 Inverse_assert ← Not(currNode.condition)
18:             else
19:                 Taken_assert ← Not(currNode.condition)
20:                 Inverse_assert ← currNode.condition
21:             end if
22:             path.conditions_smt.add(Taken_assert)
23:             path.decisions_branch.Add(indexBranchAlongPath, Result, Inverse_assert)
24:             indexBranchAlongPath ← indexBranchAlongPath + 1
25:             currNode.advance(Result)
26:         end if
27:     end while
28:     StreamOut(currPath)
29: end procedure
```

4.3. All States Once Coverage Strategy

Our proposed technique is a unified white-box testing framework for RPA workflows, based on symbolic execution, concolic execution, and a custom coverage strategy ("All States Once"), described by Algorithm 9. The framework models an RPA workflow as a control-flow graph (CFG) where each node represents an activity, and branches correspond to conditional transitions. Symbolic execution systematically explores all feasible paths by collecting constraints at each branch and solving them using the Z3 satisfiability modulo theories (SMT) solver. Concolic execution complements this by executing workflows with real input data while collecting symbolic constraints along the path, which are then mutated to explore alternate paths. The "All States Once" strategy ensures that each node in the workflow is covered at least once, optimizing test coverage with reduced cost. This technique explores all execution paths within the workflow graph G, beginning from its set of entry nodes, referred to as Initial(G). The term "alternative term" in this context refers to the distinct sets of nodes located between different paths. Let Paths(G) denote the complete set of paths within G. Any two paths, P1 and P2, in Paths(G) are considered distinct if they differ by at least one node. Notably, paths that traverse the same loop a different number

of times are not treated as separate, as the technique does not differentiate based on the number of loop iterations. Conceptually, the aim of this strategy is to achieve coverage of all workflow nodes (activities) at least once. After collecting the set of all possible paths using a depth-first search (DFS) traversal, a SMT solver—specifically, Z3—is used to generate suitable test data for each path, based on the constraints accumulated along it.

The technique was developed based on prior symbolic/concolic methods used in software testing, but adapted to the unique structure of RPA workflows, which are visually defined, data-driven, and highly dependent on UI interactions. We worked closely with industry partners, particularly UiPath, to refine the approach to be practical and efficient on real workflows. The implementation is open-source, designed with extensibility in mind, and evaluated on six diverse RPA use cases, demonstrating its ability to generate relevant test cases, achieve full coverage, and reduce manual testing effort.

Algorithm 9 Symbolic execution using the "all states once" coverage strategy.

```
procedure SymbolicExecuteAllStatesOnce(G)
    setOfPaths ← getAllPaths(G, G.entry_node, [], {}, [])
    for all P ∈ setOfPaths do
        constraints ← getPathConstraints(P)
        S ← SMTSolver.solve(constraints)
        if S is valid then
            StreamOut(S)
        end if
    end for
end procedure
function getPathConstraints(P)
    conditions_smt ← [ ]
    for i ← 0 to |P| − 2 do
        node ← P[i]
        if node.type = BRANCH then
            nextNode ← P[i + 1]
            if node.nextNode(True) = nextNode then
                evalExpected ← True
            else
                evalExpected ← False
            end if
            if evalExpected then
                conditions_smt.add(node.condition)
            else
                conditions_smt.add(Not(node.condition))
            end if
        end if
    end for
    return conditions_smt
end function
function getAllPaths(G, currNode, currPath, visitedNodes, outAllPaths)
    currPath.add(currNode)
    visitedNodes.add(currNode)
    if currNode.next is None then
        outAllPaths.append(currPath)
        return
    end if
    anyValidSuccessor ← False
    for all S ∈ currNode.next do
        if S ∉ visitedNodes then
            anyValidSuccessor ← True
            getAllPaths(G, S, currPath + {S}, visitedNodes, outAllPaths)
        end if
    end for
    if not anyValidSuccessor then
        outAllPaths.append(currPath)
    end if
end function
```

5. Evaluation and Results

We evaluate our collaborative execution architecture on real-world applications that leverage RPA processes to automate repetitive, human-performed tasks. A brief overview of each application used in the evaluation is provided below.

5.1. Bank Loan

The first application, BankLoan, is designed to verify the type of a loan based on its amount and duration. Loans under 1000 are categorized as low volume, those between 1000 and 100,000 as medium volume, and loans exceeding 100,000 as high volume. For high-volume loans, an additional step is introduced to validate a PIN associated with the bank account. This step includes a retry mechanism—if the robot fails to provide the correct PIN (e.g., the expected PIN is "1234"), the workflow will fail after a predefined number of attempts. Regardless of the volume, each loan also has a duration attribute; loans with a term shorter than five years are considered short-term, while those with longer durations are classified as long-term (see Figure 6).

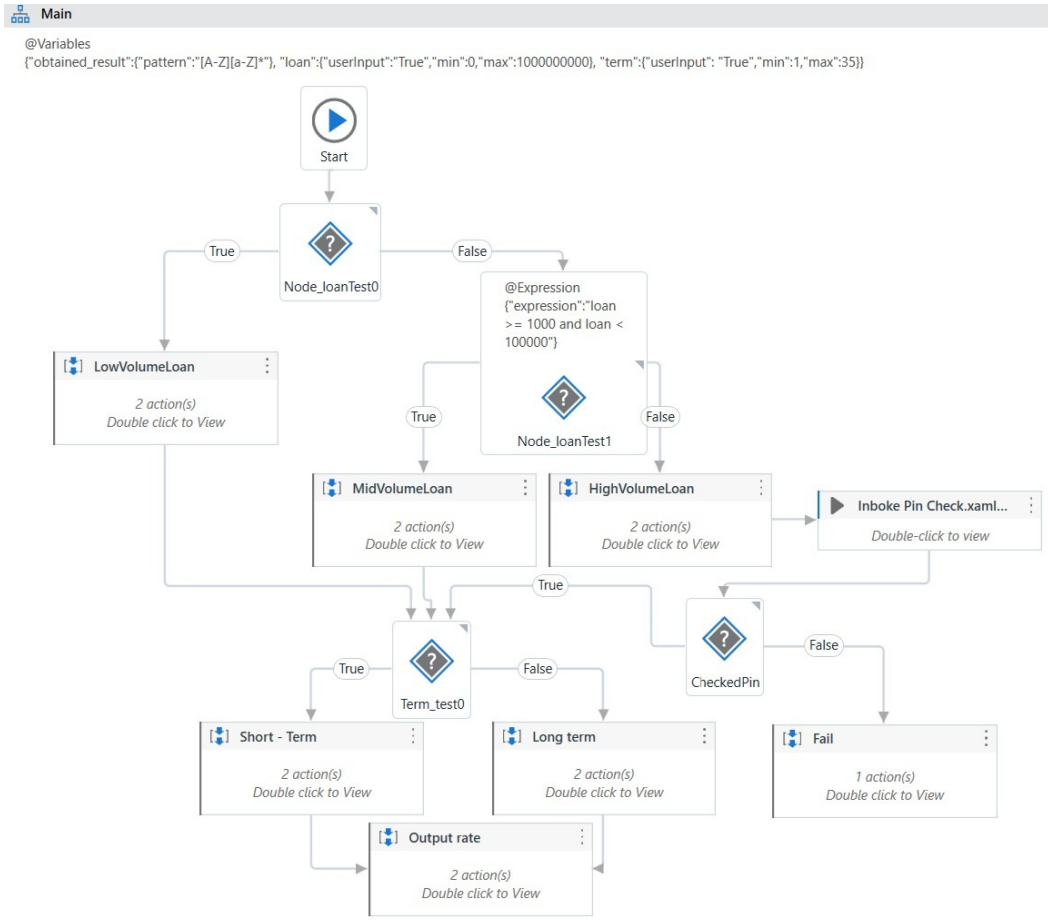

Figure 6. The main workflow of BankLoan application.

For this model, a dedicated test case was created. By executing it, testing data is generated, allowing full coverage of the model's activities. The workflow is represented by Figure 7.

In the given section, the test data for the workflow is generated by invoking the GenerateTestData.xaml workflow. This workflow begins by reading a configuration file, which contains the following parameters: the path to the Python script responsible for connecting all external components involved in the testing process, the working directory path, the Python home path. After reading the configuration, the workflow calls the Python

script, which orchestrates the external components and initiates the test data generation process. During the "when" phase, the generated test data is saved to CSV files, which will be consumed later during the execution of the workflows. In the "then" phase, the main workflow is invoked. In order to generate appropriate test data, the fuzzer component requires specific constraints for each variable. These constraints guide the fuzzer to provide input values that fall within valid intervals, match regular expressions, or represent specific types (e.g., Boolean values). To define such constraints, RPA developers can add annotations either at the workflow or activity level. In the BankLoan example, annotations are used in two main scenarios. In the first scenario, in the Main.xaml workflow, an annotation defines ranges for the main variables. For instance, the loan variable is constrained to values between 0 and 1,000,000,000, while the term variable must be between 1 and 35. These ranges allow the fuzzer to generate test data within the expected intervals. The second scenario is in the pin workflow where an annotation sets constraints for the number of retry attempts and the PIN values. The number of retries is limited to a range between 0 and 3, and the acceptable PIN values fall between 1111 and 9999. Once the annotations are processed, the generated data is saved in a CSV file named generatedTests.csv, which will be used during the testing sequence. The extracted data will be passed as parameters to the workflows during invocation.

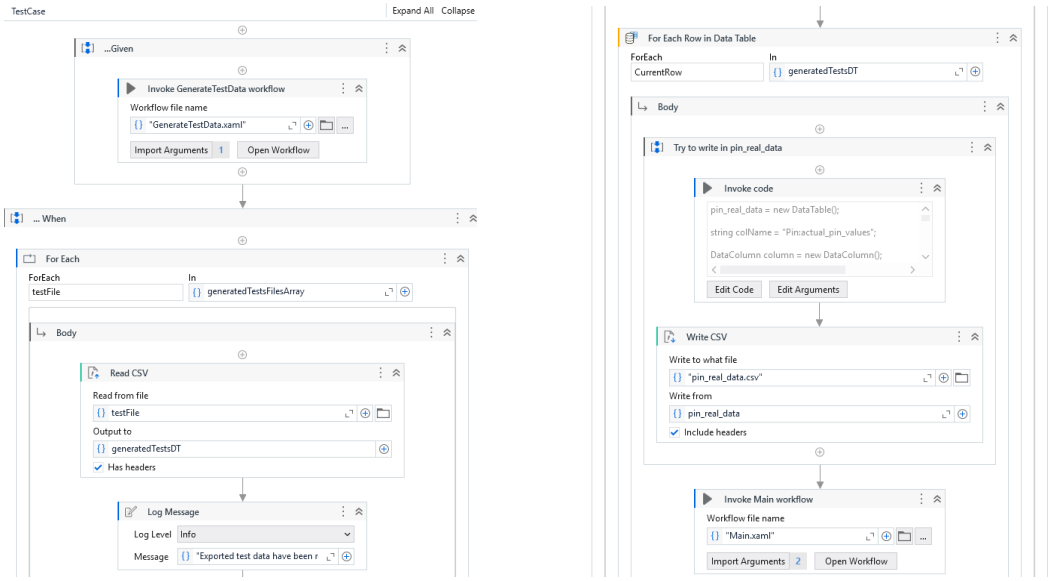

Figure 7. Test case for bank loan application.

5.2. Automating Loan Operations in Banking

Similar to the previous model, this one involves user interface interactions, according to Figure 8. It navigates to "https://uibank.uipath.com/" (accessed on 6 June 2025) to verify if a person qualifies for a loan with UiBank—a prototype website developed by UiPath Academy to demonstrate examples. Upon accessing the UiBank homepage, the robot clicks the "Products" button located in the upper-right corner, then selects "Loans" from the dropdown menu. On the loans page, the robot clicks the "Apply for a Loan" button. On the following page, several values are entered, including the applicant's email address, the loan amount requested, the loan term, the applicant's current year income, the applicant's age. If these values fall within certain predefined intervals, such as a minimum income required for the loan and an age of at least 18 years, the applicant qualifies to apply for the loan. If any of the conditions are not met, the individual is deemed ineligible to apply and must wait until all criteria are satisfied. The test data generated for this model includes the loan amount requested and whether or not the applicant is qualified for the loan.

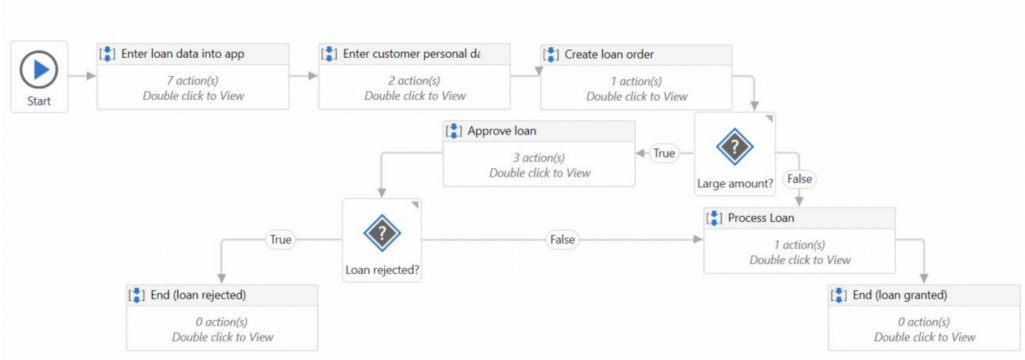

Figure 8. An example of RPA workflow that automates the loan operations for a banking process.

5.3. Application for Employee Management

In this application, the operations that can be performed are determined by the role of the user. The workflow interacts with a database containing employee details, including their name, salary, registration date, academic qualifications (specifically the presence of a master's degree), contract type, project assignment, leave requests, and notice periods. Administrators are permitted to modify the salaries of employees who have been with the organization for over one year and possess a master's degree. Conversely, users with human resources (HR) privileges can review and approve or deny employee leave requests. Leave requests are not granted if, for example, an employee with a short-term contract applies for unpaid leave or if the employee's notice period is less than five days. Furthermore, the workflow calculates the number of employees holding a master's degree and the distribution of employees across different projects. If the count of employees with a master's degree is less than or equal to two, the employer is encouraged to foster further educational development. Additionally, if fewer than two employees are assigned to a project, the workflow recommends the assignment of additional staff to the project. The workflow is represented by Figure 9.

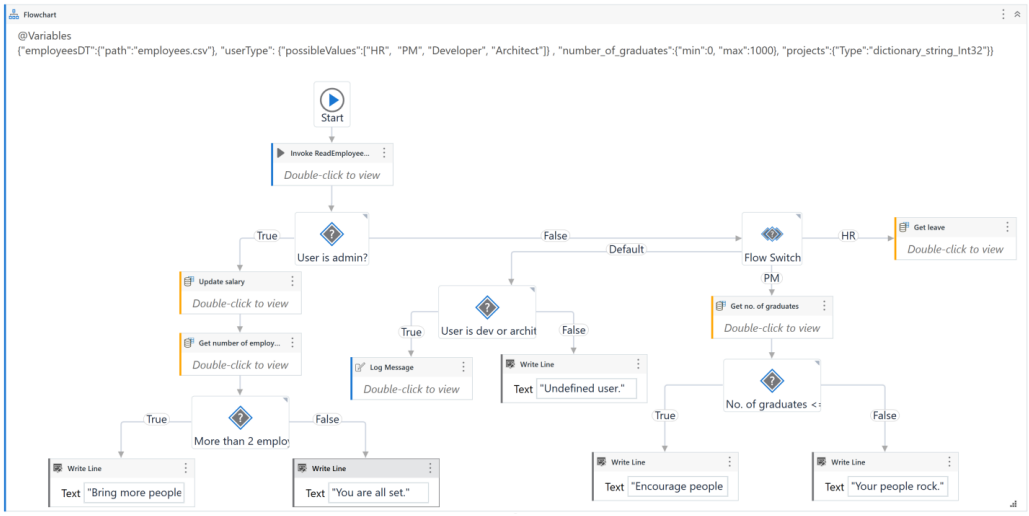

Figure 9. The main workflow of employee application.

A test case for this model is designed to generate test data that ensures comprehensive coverage of the workflow. The generated data include the type of user, the number of employees with a graduate degree, and the number of employees assigned to each project. This data allows for the evaluation of various conditions and actions within the workflow, ensuring that all potential scenarios are adequately tested.

5.4. A Hospital Services Application

The PrivateHospital model, illustrated in Figure 10, allows the user to select medical services and calculates the final price, incorporating discounts on specific services when applicable. Upon execution, the robot greets the user and requests their age. A list of available medical services is presented, and the user can select their preferred service. After each selection, the robot displays the price of the chosen service and prompts the user to decide whether they wish to add another service. If the user opts to add another service, the process repeats, and the price of the next service is added to the total cost. When the user decides to stop adding services, the final price is displayed, and the process concludes. At the outset, the user's age is requested to determine applicable discounts. Users under 18 years of age receive a 50% discount on selected services, while those over 65 years of age receive a 60% discount. Users between 18 and 65 years old are not eligible for any discounts. The generated data for a test case includes the user's age, the service selected from the input dialog (such as magnetic resonance imaging (MRI), neurosurgery consultation, or physical therapy), and the user's choice to add another service or not.

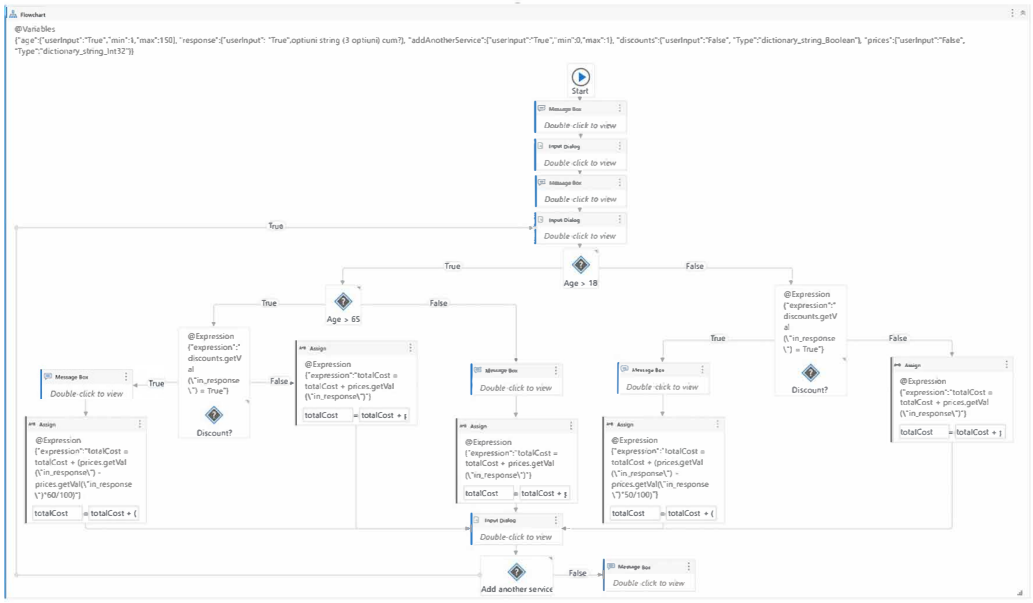

Figure 10. The main workflow of hospital application.

5.5. A Producer–Consumer Communication Pattern

The Producer and Consumer models are two distinct processes that interact through a queue in the Orchestrator, a web application designed to orchestrate the execution of repetitive business processes by facilitating the creation, monitoring, scheduling, and control of automated bots and workflows. The Producer model scans a local folder on the user's machine for PDF (Portable Document Format) files, which are invoices. It extracts key data such as client information, invoice number, due date, unit price, discount, total amount, and value added tax (VAT) number. The VAT number is determined by checking the country code from the invoice to apply the correct VAT value. If the payment due date exceeds 15 days, the invoice data is not added to the Orchestrator queue. Before inserting the data into the queue, the robot also checks a database to determine if the client has any unpaid invoices. If unpaid invoices exist, the robot includes the value of each outstanding invoice along with a penalty of 0.3% for each day the payment is overdue. The Consumer model reads the elements from the queue, processes them, and outputs the results into an Excel file. A test case for the Producer model can generate data for the payment due date, whether the vendor has unpaid invoices, the number of overdue days, and the country

code. A test case for the Consumer model can generate data for the "ShouldStop" flag provided by the Orchestrator, which is a Boolean value indicating whether the robot has been stopped by a user. When the flag is set to true, the Consumer should halt processing. Additionally, the test case can generate data for the number of transactions remaining in the queue and for the fields extracted from the invoices, such as vendor, invoice number, due date, unit price, discount, and total amount.

5.6. Spreadsheets _StateMachine—State Machine Data Processing Paradigm

The Spreadsheets_StateMachine is a model designed to process employee data from an Excel file. It reads the file, filters the data based on the salary column, and writes the filtered information into a new Excel file. The new file contains only the employees whose salary is less than 10,000. This model utilizes state machines to manage the processing steps, as indicated by its name. The state machine framework facilitates the structured flow of operations, ensuring that the data is efficiently filtered and stored according to the specified condition.

6. Metrics and Discussion

Table 1 presents various metrics reflecting the complexity of the applications under examination. These metrics include the total number of nodes (activities) and the number of branch points for each model. It is important to mention that connections within the applications that lead to cycles are considered to have an initial end node, which is counted as a branch point. This approach allows for a more comprehensive assessment of the complexity of each application's workflow, considering both linear and cyclic paths.

Table 1. The number of nodes (workflow activities) and branch points for each evaluated application.

Model Tested	Number of Nodes	Branch Nodes
BankLoanCheck	32	6
BankLoanRequest	20	11
EmployeesManagement	23	8
PrivateHospital	42	14
InvoicesProcessing	32	5
InvoicesManagementApp	39	8
Spreadsheets_StateMachine	8	3

The following two tables, Tables 2 and 3, present the metrics associated with the time required for each application to achieve 100% coverage, as well as the coverage obtained after running the fuzzer for the All states once strategy. The first table details the time performance for achieving complete coverage, while the second table outlines the extent of coverage attained by the fuzzer after processing all states in each application.

Table 2. The time needed for each application to obtain a 100% coverage.

Model Tested	Coverage Time in Minutes All States Once
BankLoanRequest	25.4 min
BankLoanCheck	18.33 min
EmployeesManagement	58.2 min
PrivateHospital	78.9 min
InvoicesProcessing	37.3 min
InvoicesManagementApp	70.2 min
SpreadsheetsStateMachine	20.4 min

Table 3. Coverage of the evaluated applications obtained after letting the fuzzer run for a fixed time of 60 min using the All states once execution strategy.

Model Tested for All States Once	Coverage for All States Once (%)
Bank loan	100%
BankLoanRequest	100%
EmployeesManagement	100%
PrivateHospital	100%
InvoicesProcessing	100%
InvoicesManagementApp	100%
SpreadsheetsStateMachine	100%

6.1. Metrics and Discussion for Concolic Execution

Table 4 presents the time required for the concolic technique to achieve full coverage of both activities (nodes) and branches. An important observation is that the time required to achieve full coverage is not directly proportional to the number of nodes, as demonstrated in the case of the InvoicesProcessing application.

To meet the demands of today's agile development processes, which include evaluating local changes before submission, continuous integration, and smoke testing, it is essential for the testing tool to provide adequate coverage within a short period of time. In order to assess this criterion, we employ the concolic execution technique to determine the depth of the testing process. The results obtained after executing the technique for one hour are presented in Table 5. This table provides an insight into how much coverage is achieved in terms of both activities and branches within the specified time frame.

Table 4. The time taken for each application to achieve 100% coverage.

Model Tested—Concolic	Coverage Time in Minutes Concolic
BankLoanCheck	33.91
BankLoanRequest	24.19
EmployeesManagement	57.30
PrivateHospital	81.43
InvoicesProcessing	45.72

Table 5. Coverage of the evaluated applications obtained after letting the fuzzer run for a fixed time of 60 min using the Concolic execution strategy.

Model Tested—Concolic	Coverage (%) Concolic
Bank loan	100%
BankLoanRequest	100%
EmployeesManagement	100%
PrivateHospital	92%
InvoicesProcessing	100%

6.2. Metrics and Discussion for Symbolic Execution

Table 6 displays the time required by the fuzzer to complete full coverage of the activities (nodes) and branches. We save the evaluation of the coverage of pairs of state values for future work in the next release of our fuzzing framework. One remark can be made at this point; the time necessary to obtain the full coverage stated is not directly related to the number of nodes. As an example, the time required to obtain full coverage for the InvoicesProcessing and InvoiceManagementApp models is substantially different, despite the underlying graphs being almost the same complexity. The fundamental reason

for this is that the later component requires particular input ranges and patterns supplied by a producer component to activate some branch points at the consumer component.

On the other hand, as we have seen in practice, obtaining strong metric coverage in a short period of time is critical for meeting today's requirements in agile development processes, such as continuous integration processes, smoke testing, evaluating local modifications before submission, and so on. To analyze the provided set of applications and our fuzzing framework in light of these requirements, we run the fuzzer on each application to determine how deep the testing process goes. Table 7 summarizes the findings.

Table 6. The time needed for each application to obtain a 100% coverage.

Model Tested—Symbolic	Coverage Time in Minutes Symbolic (%)
BankLoanRequest	30.23 min
BankLoanCheck	15.44 min
EmployeesManagement	59.4 min
PrivateHospital	73.10 min
InvoicesProcessing	29.14 min

Table 7. Coverage of the evaluated applications obtained after letting the fuzzer run for a fixed time of 60 min using the Symbolic execution strategy.

Model Tested—Symbolic	Coverage (%) Symbolic
Bank loan	100%
BankLoanRequest	100%
EmployeesManagement	69%
PrivateHospital	93%
InvoicesProcessing	100%

6.3. Complexity Comparison

Understanding the complexity of each symbolic execution strategy is important for choosing the right one for your goals. Complexity analysis helps estimate how long each algorithm might take and how much memory it might use. In Table 8, we compare the main algorithms in terms of their use of SMT solvers, execution time, and memory requirements. This comparison is based on the structure of the program being analyzed and the number of possible paths it can generate.

We use these terms:

- n: number of nodes in the program;
- b: average number of branches at a decision point;
- d: maximum depth of a path;
- S: cost of one SMT solver call;
- E: cost of executing a program node.

Table 8. Complexity analysis of symbolic execution algorithms based on time and memory.

Algorithm	Time	Memory
DFS Symbolic	$O(b^d \cdot d \cdot S)$	$O(d)$
Path Executor	$O(d \cdot (E + S))$	$O(d)$
Concolic	$O(d \cdot E)$	$O(d)$
All Paths Once	$O(b^d \cdot (d + S))$	$O(b^d \cdot d)$

6.4. Testing Recommendations Derived from Experimental Results

The results presented in the previous subsections provide valuable insights into the behavior of different testing strategies when applied to RPA workflows of varying complexity. Based on the analysis of node/branch structure, coverage time, and effectiveness under time constraints, we derive practical recommendations for RPA testing in Table 9. These are intended to support testers, tool developers, and process designers in optimizing test strategies for real-world automation workflows.

Table 9. Practical recommendations for RPA workflow testing derived from the experimental analysis.

Recommendation	Supporting Observation	When to Apply
Analyze branching structure early	Models with higher branch counts (e.g., PrivateHospital, BankLoanRequest) required significantly more time to reach full coverage.	During test planning or model inspection to allocate sufficient time and resources.
Use time-bounded fuzzing when model complexity is unknown	All models achieved 100% coverage with the "All States Once" strategy within 60 min.	In CI/CD pipelines or exploratory testing with limited time budgets.
Do not assume node count predicts test effort	InvoicesProcessing had fewer nodes yet longer coverage times due to input-dependent branches.	When estimating effort or prioritizing automation among workflows.
Symbolic and concolic techniques benefit from input-constraint modelling	Lower symbolic coverage for EmployeesManagement and PrivateHospital traced to unexercised input-dependent branches.	Whenever workflows rely on user data or dynamic behavior, integrate constraint-aware fuzzing.
Combine multiple testing strategies for comprehensive coverage	Concolic and symbolic testing explored deeper branches but with variable performance.	In complex workflows requiring high assurance, combine fuzzing with static analysis or symbolic engines.
Adopt short test iterations to support agile practices	All strategies provided strong coverage within a 60 min window, aligning with agile demands.	For fast feedback loops in development, smoke testing, and continuous integration.

These recommendations reinforce the importance of tailoring test strategies to the structural characteristics of the workflow models and the requirements of the testing environment. By leveraging both structural insights and empirical performance data, RPA testing can become more targeted, scalable, and aligned with modern development practices.

7. Conclusions and Future Work

This paper presented a unified framework for automated testing of RPA workflows using symbolic and concolic execution strategies. The approach models RPA processes as control-flow graphs and leverages SMT-based path exploration to generate high-coverage test suites. A novel "All States Once" coverage strategy was introduced to ensure practical completeness in node coverage with reduced computational overhead.

Through empirical evaluation on six real-world RPA applications—including banking workflows, invoice processing, and hospital services—we demonstrated that the proposed method achieves full node and branch coverage in all tested cases, with execution times ranging from 18.33 min to 78.9 min, depending on workflow complexity. Our results confirm that symbolic execution is better suited for smaller workflows requiring full path precision, while concolic execution scales better to broader coverage in time-constrained scenarios.

We also showed that annotations play a critical role in narrowing the input space and avoiding infeasible paths, improving solver efficiency and test relevance. The ability to mock APIs and inject domain-specific constraints made the framework adaptable to industrial use cases.

Building on the advancements presented in this study and informed by gaps identified in the current literature, several directions for future research emerge. The framework can be extended to accommodate more complex and dynamic RPA workflows. Our results raise questions around how to scale the execution strategies to very large workflows, and how to integrate learning-based techniques to improve path prediction and constraint solving in complex business environments. Machine learning can also be incorporated for automated test case generation, while prioritization holds significant potential to increase testing efficiency and further reduce manual efforts. One option is the automated selection of symbolic and concolic strategies, based on workflow characteristics such as depth, branching factor, and input space complexity. Another promising direction is to extend the framework to support data-driven RPA workflows, where control decisions depend on unstructured or external data sources. In addition, RPA-specific fault models could enhance the precision of test generation, especially in exception handling paths. These open research questions highlight important challenges that warrant further investigation to advance the state of automated testing in RPA.

We would like to express our gratitude to UiPath for their collaboration and support throughout the research and development of the proposed methods. This research was partially supported by the project "Romanian Hub for Artificial Intelligence - HRIA", Smart Growth, Digitization and Financial Instruments Program, 2021-2027, MySMIS no. 334906.

Author Contributions: Conceptualization, C.P., M.C. and A.-N.S.; methodology, C.P., M.C. and A.-N.S.; software, C.P., M.C. and A.-N.S.; validation, C.P., M.C. and A.-N.S.; formal analysis, C.P., M.C. and A.-N.S.; investigation, C.P., M.C. and A.-N.S.; resources, C.P., M.C. and A.-N.S.; data curation, C.P., M.C. and A.-N.S.; writing—original draft preparation, C.P., M.C. and A.-N.S.; writing—review and editing, C.P., M.C. and A.-N.S.; visualization, C.P., M.C. and A.-N.S.; supervision, C.P., M.C. and A.-N.S.; project administration, C.P., M.C. and A.-N.S.; funding acquisition, C.P., M.C. and A.-N.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by the project "Romanian Hub for Artificial Intelligence—HRIA", Smart Growth, Digitization and Financial Instruments Program, 2021–2027, MySMIS no. 334906.

Data Availability Statement: The data and source code presented in this study are openly available at: https://github.com/unibuc-cs/rpa-testing (accessed on 6 June 2025).

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Gartner. *Magic Quadrant for Robotic Process Automation Software*; Market Research Report, No. G00379618; Gartner Research: Stamford, CT, USA, 2019.
2. Gartner. *Magic Quadrant for Robotic Process Automation*; No. G00800237; Gartner Research: Stamford, CT, USA, 2024.

3. Cernat, M.; Staicu, A.; Stefanescu, A. Towards Automated Testing of RPA Implementations. In Proceedings of the 11th International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST), Virtual, 8–9 November 2020; ACM: New York, NY, USA, 2020; pp. 21–24.
4. Chacón Montero, J.; Jimenez Ramirez, A.; Gonzalez Enríquez, J. Towards a Method for Automated Testing in Robotic Process Automation Projects. In Proceedings of the AST'19 Workshop, Hamburg, Germany, 19–20 February 2019; pp. 42–47.
5. Holmberg, M.; Dobslaw, F. An Industrial Case Study on GUI Testing with RPA. In Proceedings of the 2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Valencia, Spain, 4–13 April 2022; pp. 199–206.
6. Yatskiv, S.; Voytyuk, I.; Yatskiv, N.; Kushnir, O.; Trufanova, Y.; Panasyuk, V. Improved Method of Software Automation Testing Based on the Robotic Process Automation Technology. In Proceedings of the 9th International Conference on Advanced Computer Information Technologies (ACIT), Ceske Budejovice, Czech Republic, 5–7 June 2019; pp. 293–296.
7. Cernat, M.; Staicu, A.; Stefanescu, A. Improving UI Test Automation Using Robotic Process Automation. In Proceedings of the 15th International Conference on Software Technologies (ICSOFT), Paris, France, 7–9 July 2020; ScitePress: Setúbal, Portugal, 2020; pp. 260–267.
8. Zhang, Z.; Chen, P.; Xie, T. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle, WA, USA, 17–21 July 2023; pp. 1–12.
9. Alshmrany, K.M.; Menezes, R.S.; Gadelha, M.R.; Cordeiro, L.C. FuSeBMC: A White-Box Fuzzer for Finding Security Vulnerabilities in C Programs. *arXiv* **2020**, arXiv:2012.11223.
10. Jaffar, J.; Maghareh, R.; Godboley, S.; Ha, X.L. TracerX: Dynamic Symbolic Execution with Interpolation. *arXiv* **2020**, arXiv:2012.00556.
11. Bardin, S.; Delahaye, M.; Kosmatov, N.; Laleau, R.; Le Gall, P. A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *Int. J. Softw. Tools Technol. Transf.* **2012**, *15*, 1–15.
12. Chen, B.; Havlicek, C.; Yang, Z.; Cong, K.; Kannavara, R.; Xie, F. CRETE: A Versatile Binary-Level Concolic Testing Framework. In Proceedings of the 21st International Conference on Fundamental Approaches to Software Engineering (FASE), Thessaloniki, Greece, 14–18 April 2018; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2018; Volume 10802, pp. 281–298.
13. Paduraru, C.; Staicu, A.N.; Stefanescu, A. Robotic Process Automation for the Gaming Industry. In Proceedings of the 18th International Conference on Software Technologies (ICSOFT), Rome, Italy, 10–12 July 2023.
14. Khankhoje, R. Robotic Process Automation (RPA) Towards Automation Testing. *Int. J. Adv. Res. Comput. Sci.* **2023**, *15*, 1930–1936. [CrossRef]
15. Yu, S.; Fang, C.; Tuo, Z.; Zhang, Q.; Chen, C.; Chen, Z.; Su, Z. Vision-Based Mobile App GUI Testing: A Survey. *arXiv* **2023**, arXiv:2310.13518.
16. Wickramasinghe, M.; Perera, I. Automating UI Issue Detection Using Convolutional Neural Networks and Machine Learning Models. In Proceedings of the 9th International Conference on Multidisciplinary Research, Colombo, Sri Lanka, 7–9 November 2024; pp. 1–6.
17. Wang, W.; Yang, W.; Xu, T.; Xie, T. VET: Identifying and Avoiding UI Exploration Tarpits. *arXiv* **2021**, arXiv:2102.06377.
18. Ravi, A. Exploring RPA (Robotic Process Automation) as a Means to Test and Develop User Interfaces. In *Purdue University Technical Report*; Submission for CNIT 58100-047—Fall; Purdue University: West Lafayette, IN, USA, 2021.
19. Baldoni, R.; Coppa, E.; D'Elia, D.C.; Demetrescu, C.; Finocchi, I. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* **2018**, *51*, 50. [CrossRef]
20. De Moura, L.; Bjørner, N. Z3: An Efficient SMT Solver. In Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary, 29 March–6 April 2008; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2008; Volume 4963, pp. 337–340.
21. Godefroid, P.; Levin, M.Y.; Molnar, D.A. SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM* **2012**, *55*, 40–44. [CrossRef]