RPA Testing Using Symbolic Execution

Ciprian Paduraru, Marina Cernat and Adelina-Nicoleta Staicu
Department of Computer Science, University of Bucharest, Romania

Keywords: RPA, Testing, Symbolic Execution, Test Generation.

Abstract: The goal of Robotic Process Automation (RPA) technology is to identify patterns in repetitive processes that can be automated in enterprise workflows, and to create intelligent agents that can repeat those processes contextually and without human effort. However, as the technology has evolved considerably in terms of model complexity, data inputs, and output dimensionality, preserving the quality of the building blocks of the operations is a difficult task. We identified that there is a gap in testing methods and tools capable of efficiently testing RPA workflows. In this paper, we therefore propose to address this gap by using symbolic execution as a starting point. We focus on both the methodology and algorithms required to transfer existing research in symbolic execution to the RPA domain, propose a tool that can be used by researchers and industry, and present our current evaluation results for various use cases along with best practices.

1 INTRODUCTION

Robotic Process Automation (RPA) (Aguirre and Rodriguez, 2017), (Leno et al., 2021) aims to reduce the workload of repetitive tasks performed by employees in organizations. Rather than intervening in the information system itself, as is the case with traditional workflow optimization, RPA generally works by identifying patterns in repetitive processes that can be automated. The identified patterns are written in a specific language that we refer to as *RPA workflow* so that a software robot can perform those operations. An *RPA workflow* consists of the identified activities and the relationships between them. Thus, the underlying abstract format of these activities takes the form of a graph in which each activity is a node (or more precisely a hyper-node with multiple nodes). Overall, the purpose of RPA workflows is to enable both technical and non-technical people to create and manage content. The operations performed as part of the activities generally relate to interactions with user interfaces (Cernat et al., 2020a), cross-enterprise functionality such as database or server management, or custom APIs that connect to client infrastructure. An *RPA robot* is an entity that executes workflows and replaces repetitive human activities. It executes a workflow by processing the activities and operations specified in the nodes.

Recently, RPA is considered by many companies as one of the most efficient ways to reduce costs and achieve a high return on investment (ROI) (Ray and et al., 2022), (Economy, 2020). RPA is being used in industries such as manufacturing, technology and healthcare, across a range of business sizes, from small businesses to large enterprises. As RPA technology takes hold and workflows created in enterprises automate and connect many endpoints, it is important to test these automation processes from start to finish. In general, RPA testing means defining inputs and expected outcomes and verifying that the automation process works as expected. An automated testing tool for RPA can detect common problems such as logic errors, security vulnerabilities, or other low-level coding issues.

The goal of our work is to provide a framework for testing RPA workflows and to identify best practices based on the evaluation of the results. The work was done in collaboration with our industry partner UiPath[1], which is currently the world-leading RPA technology company (according to (Ray and et al., 2022)).

Based on our literature research, our framework brings the following innovations to the field:

- To the best of our knowledge, this is the first academic work describing a tool capable of testing an RPA workflow from start to finish.
- We adapt existing modern symbolic execution methods to the RPA case.

[1]https://uipath.com

Paduraru, C., Cernat, M. and Staicu, A.
RPA Testing Using Symbolic Execution.
DOI: 10.5220/0012061000003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 269-275
ISBN: 978-989-758-665-1; ISSN: 2184-2833

- We identify best practices for testing RPA workflows from our collaboration with industry partners by applying our methods to real-world applications.

Note that more technical details on our tool are provided in our complementary paper (Paduraru et al., 2023), which is based on a common technical foundation but provides a concolic approach, not the symbolic approach presented in here.

The rest of the paper is organized as follows. Section 2 discusses previous work in the field of RPA and software testing. Furthermore, Section 3 provides an overview of the testing pipeline. Section 4 discusses how symbolic execution methods are used by our framework in the RPA testing environment. The evaluation of the results are presented in section 5. The last section is dedicated to conclusions.

2 RELATED WORK

RPA and Automated Testing. By reviewing the literature on RPA and testing, we concluded that there is a lack (from the authors understanding, because there is no prior work at all) in white-box testing for the RPA workflows. In (Cernat et al., 2020b), the authors describe the state of the practice for testing software robots and propose some ideas for test automation using model-based testing, without providing any further evaluation or implementation proposal.

The strategy employed by (Chacón Montero et al., 2019) is to set up a testing environment by watching how people interact with application UI interfaces and documenting their actions with logs and pictures. Firstly, because the method does not evaluate an RPA workflow, it is more appropriate to classify it as "black-box testing". It has limitations compared to our proposed methods in two ways, despite being interesting for our efforts: (a) it is more difficult to introduce user knowledge into the test field, and (b) using only computer vision approaches to record operations can significantly limit the coverage of the test state space compared to white-box testing of a given graph. Secondly, RPA was utilized in the publications (Holmberg and Dobslaw, 2022) and (Cernat et al., 2020b) to test graphical user interfaces. Additionally, RPA has been used in (Yatskiv et al., 2019) for automation and regression testing for additional common information system components, like web, desktop, or mobile applications, or even API functionality testing in large-scale software.

Testing Methods. Our research focuses on white-box testing (Godefroid et al., 2012a) using symbolic execution approaches. We utilize current state-of-the-art techniques, but our contribution is to investigate and assess the delta of modifications for testing RPA workflows. For testing at the binary, source code, or LLVM level, we have modified our procedures to fit existing solutions. In terms of techniques and algorithms for symbolic execution, our work was influenced by two popular open source frameworks: KLEE (Bucur et al., 2011) and S2E (Chipounov et al., 2012). We drew inspiration from tools like CRETE (Chen et al., 2018) for testing at the LLVM representation level, which can be somewhat analogous to our notion of *annotation*.

3 TESTING PIPELINE OVERVIEW

This section gives an overview of the involved components, pipelines, and user sides of the proposed automated RPA testing framework.

The process in Fig. 1 presents the components of the testing tool and their relations during the execution. The implemented components are:

- XAML parser, which is a transformer from the robot which is provided in its native XAML format to an annotated graph format, which is suitable for the testing tool;
- symbolic test generator using also fuzzing;
- integrating script (in Python), which connects the above components.

The integrating script gets as input a file *userInput.json* containing the configuration for the XAML parser and the test generator. It must be provided by the user at the beginning of the testing process, as a prerequisite. The purpose of the integrating script, as its name states, is to connect the two components, the XAML parser and the test generator. The the XAML parser is used to parse XAML files, the result being serialized in a .json file, which preserves only the information necessary to run the symbolic test generator. The execution of both components is performed with the help of the integrating script having as result the following files:

- *outputXamlParser.json*, representing the output of the XAML parser;
- *generatedTests.csv*, representing the output of the symbolic test generator;
- *executionLogs.csv*, with the purpose of giving details about the execution of the symbolic test generator such as the solver strategy used and the execution time;

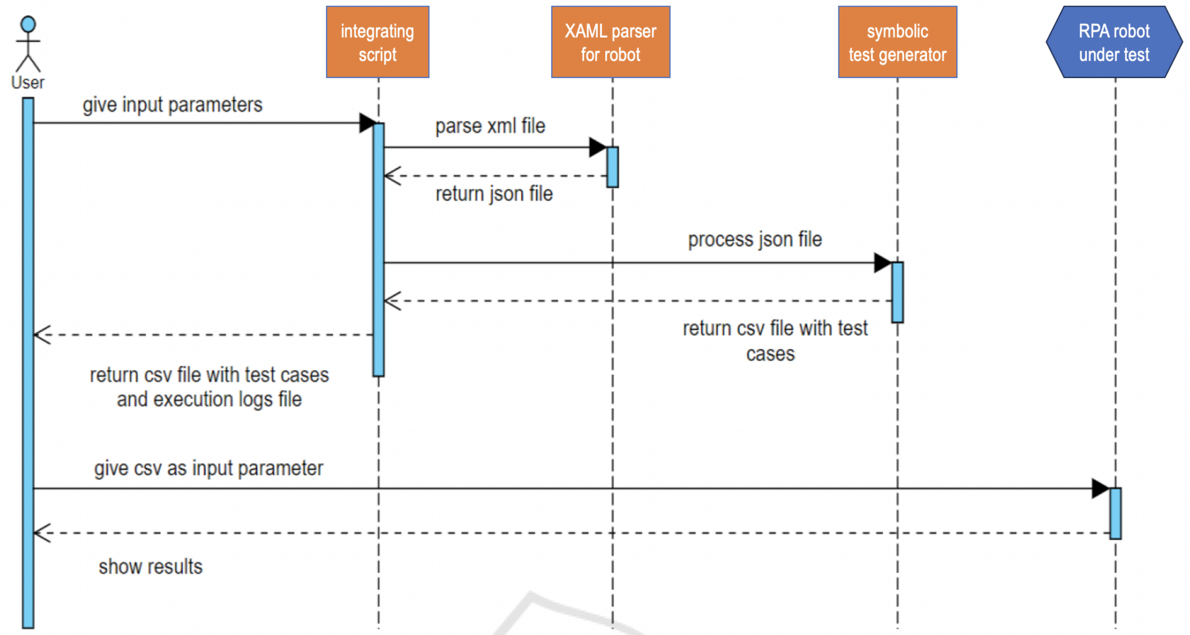

Figure 1: The overview of the testing process.

- *CFGraph.png*, representing the control flow graph of the robot.

In addition to these components, we have created an extra reusable RPA robot designed to execute the integrating script and return an array of test files generated by the symbolic test generator. The resulted test suite can be run within the test execution framework provided by the RPA tool. The coverage metrics can be extracted from there.

4 SYMBOLIC EXECUTION STRATEGY FOR TESTING RPA WORKFLOWS

In this section, we describe our proposal for applying the symbolic execution methodology to the RPA domain, detailing both the methods and the implementation level. Our contribution in this sense is to incorporate them into our framework design and provide our considerations and requirements needed for testing RPA workflows. An important advantage of testing RPA workflows is that the entire model, consisting of activities and connections between them, is known. With the help of the human in the loop, who can additionally provide what we identified earlier in the text as *annotations*, the test process turns out to be computationally suitable in most cases, even for methods known to be very computationally intensive in other use cases, such as symbolic execution. The test corpus might then have the advantage of completeness, while also being sustainable from a computational effort or time horizon perspective.

The methods described below formally assume that the RPA workflow takes the structure of a directed graph G in which each activity is a node. Cycles can exist in this graph, and the starting activity (i.e., node) is denoted $Initial(G)$. The decision factor for a test case generation and execution process is mainly related to branch points, i.e., when the next activities are different depending on a certain condition. The general idea is that when a branch point B occurs on a path P of activities within G, the condition on which the branch depends can be solved symbolically with an SMT solver to check its feasibility. In our case, we chose Z3 (Mendona de Moura and Bjorner, 2008) as the SMT solver for the framework. Several strategies can be found in the literature, depending on the complexity of the RPA workflow. In practice and in collaboration with our industrial partner, we found that most of the workflows used in the production phase are small to medium in size. The symbolic execution method (Baldoni et al., 2018) exhaustively explores the graph G including loops. Since the graphs of the workflows are usually small to medium in terms of the number of operations (they generally need to be evaluated visually), we have empirically found that this method offers the best tradeoff between the completeness of the generated test corpus and the computational efficiency. However, there are rare cases where the workflows are very large, and in these cases concolic evaluation might have an advantage as it covers the activities per unit time faster, cf. (Paduraru et al., 2023).

Listing 1 shows the main loop of the implementation pseudocode in our framework. The process starts with the initial activity node in the graph G (Line 2), inserts this initial path with a single node into a worklist W, then extracts a promising path at each step of the loop (with several possible customizable strategies for evaluating path priorities), and continues its forward execution, Line 7. The worklist W is a priority queue sorted by a priority of elements (paths). By default, this priority is calculated based on how deep in the G exploration tree the path is now. By default, paths that are at the top of the exploration tree are promoted so that the algorithm is able to traverse a variety of paths and activities in less time, rather than focusing on single long paths with common activity nodes. However, depending on the test goal, the user is free to override this default behavior by using function hooks in our framework.

When generating new inputs to the work queue W, the default calculation method for setting their priority is based on the location (depth index) in the tree where the branching constraint is reversed. Thus, when sorting the priority queue W by priority, the default behavior is to promote changes to paths at the beginning of previous paths. This is also used in (Godefroid et al., 2012b), but the user is free to override this functionality with a hook function as needed.

```
1 // Init the start exploration path with the entry node of
        the workflow graph
2 start_path = SMTPath([G.entry_node], priority =
       MAX_PRIORITY)
3 // Add this to the current Worklist
4 W.add(start_path)
5 while W.notEmpty():
6   currPath = W.extract()
7   ExecutePath(currPath, W)
```

Listing 1: The main loop of Symbolic execution using DFS strategy

The code for continuing the execution of a given path is shown in Listing 2. An execution path in this purely symbolic model is abstracted by the code of class *SMTPath*. Given knowledge of an initial set of constraints on the variables in the DataStore (e.g., those propagated from annotations), Line 6 adds them to the state of the SMT solver. Next, the implementation parses the current node to make the continuation decision, Line 9.

When a branch node occurs, Lines 15-41, the current path is cloned and split in two: one takes the *True* branch and the other takes the *False* branch. We add only the feasible paths to the future worklist after calculating their priority and setting up the next node for execution according to the considered branch. It is important to note that for one of the paths, the DataStore instance is also duplicated, Line 38. This ensures that both operate on different data context values. This is also beneficial for parallelization, as branches from the worklist can then be executed in parallel. Note that the implementation presented is depth-first search (DFS) style, as the next highest priority path continuation is the currently actively executing path, while the other is added to the worklist and possibly executed later in the process, Line 40.

```
1 ExecutePath(path : SMTPath, W : Worklist)
2   // Each path stores the next node to execute (initial
       is the entry node of the path)
3   currNode = path.nextNodeToExecute
4
5   // First, add all the constraints given by the
       DataStore objects restrictions given by annotations
6   path.conditions_smt = DataStoreTemplate.
       getVariablesAssertions()
7
8   // The node will advance to the end of the Workflow
       graph G at each iteration of the path
9   while currNode != None:
10    // if current node is not a branch type, just execute
        it and advance to the next node in the workflow
11    if currNode.type != BRANCH:
12      WorkflowExecutor.Execute(currNode)
13      currNode = currNode.next()
14    // If the current node is a branch,
15    else:
16      // If branch node but there is no symbolic variable
        used in its condition expression then just advance
17      if not HasSymbolicVariables(currNode.condition )
18        // Get the result of the evaluation and advance
19        Result = WorkflowExecutor.Execute(currNode)
20        currNode.advance(Result)
21        continue
22
23      // Node has symbolic variables, build the
       continuation of the current path on both
24      // True and False branches with the next nodes and
       SMT assertions needed
25      newPath_onTrue = SMTPath(nodes=currPath.nodes +
       currNode.nextNode(True),
26                    conditions_smt= currPath.
       conditions_smt + currNode.condition)
27      newPath_onFalse = SMTPath(nodes=currPath.nodes +
       currNode.nextNode(False),
28                     conditions_smt= currPath.
       conditions_smt + Not(currNode.condition))
29      // Then check which of them are feasible,
       prioritize, set the next starting nodes, clone the
30      // data store instances such that they could work
       in parallel, and add them to the worklist
31      solvableNewPaths = []
32      if SMTSolver(newPath_onTrue) has solution:
33        solvableNewPaths.add(newPath_onTrue)
34      if SMTSolver(newPath_onFalse) has solution:
35        solvableNewPaths.add(newPath_onFalse)
36      for newPath in solvableNewPaths:
37        newPath.scorePath()
38        newPath.data_store = currPath.data_store.clone()

39        newPath.nextNodeToExecute = currNode.nextNode(
       True if newPath == newPath_onTrue else False)
40        W.add(newPath_onFalse)
41      break
42    // If at the end of the path, then stream out the
       model variables
43    StreamOut(currPath)
```

Listing 2: The main loop of symbolic execution using DFS strategy

Other implementations might also consider breadth-first-search (BFS) style, i.e., after a branch, both branches are added to the worklist according to priority, and a new path must be considered at

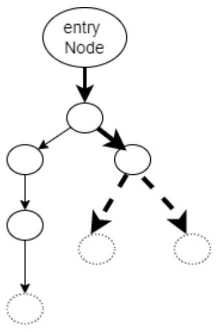

Figure 2: The figure shows the difference between the execution types DFS and BFS. Frontier nodes are represented by broken circles, while the remaining nodes are considered visited nodes. The path in bold is the current (i.e., in execution) path. In BFS-style execution, both new boundary nodes are added to the priority queue worklist, and the next iteration selects the top continuation node (by priority) - possibly even the unexplored node on the left side of the workflow graph. In DFS fashion, one of the two new nodes is added to the work queue and further execution continues on it.

each step. Figure 2 visually explains the difference between the two.

5 EVALUATION

We evaluate the framework by providing metrics and some other best practices that we identified while working on a number of real-world applications automated by RPA workflows. A brief description of these applications can be found below:

- *BankLoanCheck*: The automation of the eligibility check of customers based on features such as the amount requested, loan term, existing credit cards and loans.
- *BankLoanRequest*: The automation of the credit processing as a continuation of the above process after the credit check.
- *EmployeesManagement*: The automation of various HR processes for employees within an organization.
- *PrivateHospital*: The automation of some processes managing the relationship between a hospital and its customers.
- *InvoicesProcessing*: The automation of a producer-consumer paradigm that processes invoices in PDF format and sends them to a queue that automatically processes their inputs in a connected database and operations.

Metrics and Discussion. To get an overview of the complexity of the applications evaluated, Table 1

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

Model tested	Nodes	Branch nodes
BankLoanCheck	32	6
BankLoanRequest	20	11
EmployeesManagement	23	8
PrivateHospital	42	14
InvoicesProcessing	32	5

Table 2: The time needed (in minutes) for each application to obtain a 100% coverage.

Model tested	Time
BankLoanCheck	30.23 min
BankLoanRequest	15.44 min
EmployeesManagement	117.4 min
PrivateHospital	73.10 min
InvoicesProcessing	29.14 min

shows the number of total activity nodes and branch points for each of these applications

Also, Table 2 shows the time needed for the fuzzer to achieve full coverage of the activities (nodes) and branches. We leave the evaluation of the coverage of pairs of state values to future work in the next version of our testing framework. One observation that can be made to this point is that the time required to achieve the full coverage discussed is not directly proportional to the number of nodes. As a concrete example, the time required to achieve full coverage for the *BankLoanRequest* and *EmployeesManagement* models is very different, even though the corresponding complexity of the underlying graphs is almost the same. The main reason for this is that the latter component requires certain input ranges and patterns to be proposed by a producer component in order to trigger some branch points at the consumer component.

On the other hand, as we have observed in practice, it is important to obtain good metric coverage within a short time to meet today's requirements in agile development processes, e.g., continuous integration processes, smoke testing, evaluation of local changes before submission, etc. To evaluate the proposed set of applications and our testing framework considering these requirements, we run the tool on each application for a fixed time of 60 minutes to understand how deep the testing process goes. The results are shown in Table 3.

Other Lessons Learned Identified During the Evaluation Process. The following are additional features implemented in our framework that have been helpful in validating results and usability for industry.

- Seeds: Setting fixed seeds for the fuzzer helped

Table 3: Coverage of the evaluated applications obtained after letting the fuzzer run for a fixed time of 60 minutes.

Model tested	Coverage (%)
BankLoanCheck	100%
BankLoanRequest	100%
EmployeesManagement	69%
PrivateHospital	93%
InvoicesProcessing	100%

the testing process both in implementation and debugging, and in validating and reproducing results while problems were observed.

- Results streaming support: This greatly facilitated the evaluation process in practice.
- Results display options: Various options such as displaying the variable context even if it is not used in the branching processes, support for logging to see the path of the current state in the test process, debug images to visualize the transformed graph between the pipeline components.
- Extensible, user-friendly annotations and mocking support.

6 CONCLUSION AND FUTURE WORK

In this paper we presented a framework for generating test suites suitable for RPA workflows testing. The motivation started by the growing acceptance ratio of RPA in different sectors and ranges of organization sizes. Our main contribution was to research, observe, and adapt existing state of the art methods for the RPA testing field. The results and best practices identified could be valuable further both for academic and industry communities. We believe that this is only the starting point and there are many future ideas that can be investigated and implemented. For instance, we will apply concolic execution for larger suites of input workflows and bring in more machine learning methods to testing, such as reinforcement learning agents that can guide the testing process towards certain goals faster. Also, we would like to add more human-in-the-loop features in the testing phase, to direct better the computational efforts. In the technical field, we would like to increase the framework's chance of being adopted easier via a smoother integration of existing implemented workflows, which were not previously authored for being tested (e.g., no annotation support).

ACKNOWLEDGEMENTS

We thank UiPath for their collaboration and support during the research and development of the proposed methods. This research was supported by the European Regional Development Fund, Competitiveness Operational Program 2014-2020 through project IDBC (code SMIS 2014+: 121512).

REFERENCES

Aguirre, S. and Rodriguez, A. (2017). Automation of a business process using robotic process automation (RPA): A case study. In *Proc. of Applied Computer Sciences in Engineering - 4th Workshop on Engineering Applications*, volume 742 of *CCIS*, pages 65–71. Springer.

Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. (2018). A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39.

Bucur, S., Ureche, V., Zamfir, C., and Candea, G. (2011). Parallel symbolic execution for automated real-world software testing. In *Proc. of the 6th Conf. on Computer Systems*, EuroSys'11, pages 183–198. ACM.

Cernat, M., Staicu, A., and Stefanescu, A. (2020a). Improving UI test automation using robotic process automation. In *Proc. of the 15th Int. Conf. on Software Technologies (ICSOFT'20)*, pages 260–267. ScitePress.

Cernat, M., Staicu, A., and Stefanescu, A. (2020b). Towards automated testing of RPA implementations. In *Proc. of 11th Int. Work. on Automating TEST Case Design, Selection, and Evaluation (A-TEST'20)*, pages 21–24. ACM.

Chacón Montero, J., Jimenez Ramirez, A., and Gonzalez Enríquez, J. (2019). Towards a method for automated testing in robotic process automation projects. In *AST'19 Workshop*, pages 42–47.

Chen, B., Havlicek, C., Yang, Z., Cong, K., Kannavara, R., and Xie, F. (2018). CRETE: A versatile binary-level concolic testing framework. In *21st Int. Conf. on Fundamental Approaches to Software Engineering (FASE'18)*, volume 10802 of *LNCS*, pages 281–298. Springer.

Chipounov, V., Kuznetsov, V., and Candea, G. (2012). The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49.

Economy, A. (2020). Executive summary - the 2021 state of rpa survey report. *Report G00319864. Gartner.*

Godefroid, P., Levin, M. Y., and Molnar, D. (2012a). SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27.

Godefroid, P., Levin, M. Y., and Molnar, D. A. (2012b). SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44.

Holmberg, M. and Dobslaw, F. (2022). An industrial case-study on gui testing with rpa. In *2022 IEEE Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 199–206.

Leno, V., Polyvyanyy, A., Dumas, M., Rosa, M. L., and Maggi, F. M. (2021). Robotic process mining: Vision and challenges. *Bus. Inf. Syst. Eng.*, 63(3):301–314.

Mendona de Moura, L. and Bjorner, N. S. (2008). Z3: an efficient SMT solver. In *14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 337–340. Springer.

Paduraru, C., Cernat, M., and Staicu, A.-N. (2023). Concolic execution for RPA testing. In *27th International Conference on Engineering of Complex Computer Systems (ICECCS'23)*. IEEE, to appear.

Ray, S. and et al. (2022). Gartner magic quadrant for robotic process automation. *Report 4016876. Gartner*.

Yatskiv, S., Voytyuk, I., Yatskiv, N., Kushnir, O., Trufanova, Y., and Panasyuk, V. (2019). Improved method of software automation testing based on the robotic process automation technology. In *9th Int. Conf. on Advanced Computer Information Technologies (ACIT)*, pages 293–296.