Documentation and technical report for automated RPA testing

Authors: Ciprian Paduraru, Alin Stefanescu, Adelina Staicu, Marina Cernat, Ana Iova, Bogdan Dediu

University of Bucharest, Romania

Version 0.8 - 4th of October 2021

Version 0.9 - 18th of October 2021

Version 1 – 9th of November 2021

Table of Contents

1. End-to-end testing process

1.1 User guide. (installing and running the tool)

Step 0. Clone the following repository to your C drive and make sure to deactivate your antivirus:

https://github.com/unibuc-cs/rpa-testing

Step 1. Install Python 3.8 and the following packages:

pip install z3-solver

pip install py_expression_eval

pip install networkx

pip install pygraphviz

A note about the pygraphviz package: If there are problems with installing this package (on Windows), the best solution is to install the graphviz binaries and sources (from https://graphviz.org/download/source), then build the pygraphviz from source code for python:

https://github.com/pygraphviz/pygraphviz/blob/main/INSTALL.txt

In case that link disappears:

1. Download and install 2.46.0 for Windows 10 (64-bit):

`stable_windows_10_cmake_Release_x64_graphviz-install-2.46.0-win64.exe

<https://gitlab.com/graphviz/graphviz/-/package_files/6164164/download>`_.

2. Install PyGraphviz via

.. code-block:: console

```
  PS C:\> python -m pip install --global-option=build_ext `
              --global-option="-IC:\Program Files\Graphviz\include" `
              --global-option="-LC:\Program Files\Graphviz\lib" `
              pygraphviz
```

Step 2. Install UiPath Studio Pro version 2021.4.4 or above.

Step 3. **Access TestingToolStable directory.** The main components of the testing flow can be found in the Application directory: XMLParser, Fuzzer, C#Models. These 3 components are "connected" using integrationScript.py and userInput.json as follows:

userInput.json represents the configuration selected by the user based on the workflow under test, the parameters required by the XMLParser and Fuzzer. The paths inside the JSON file must be configured by the user.

For example:

```
{
  "xamlPath": "{PathBeforeRpaTestingFolder}\\rpa-
testing\\TestingToolStable\\Applications\\C#Models\\ContractModel\\Main.xaml",
  "modelBasePath": "{PathBeforeRpaTestingFolder}\\rpa-
testing\\TestingToolStable\\Applications\\C#Models\\ContractModel\\",
  "solverStrategy": "STRATEGY_DFS",
  "outputTests_MaxTestPerFile": 5,
  "debug_tests_fullPaths": 0,
  "debug_ConsoleOutput": 0,
  "debug_tests_fullVariablesContent": 0,
  "seedsFile": null,
  "numRandomGeneratedSeeds": 0
}
```

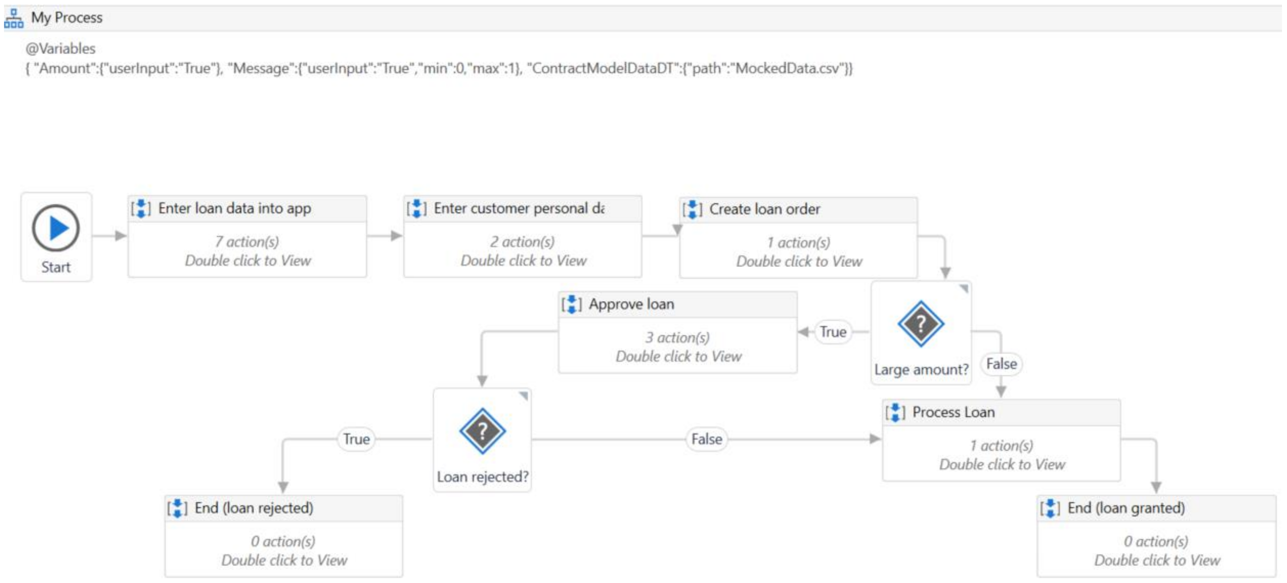
This configuration is used to generate test cases for Main.xaml workflow using STRATEGY_DFS. Each parameter can be edited by the user before running the testing tool.

Note: for more details on each parameter, see Section 3.

Step 4. Access TestingToolStable \Applications\C#Models directory containing 2 model examples: ContractModel and SimpleBankLoanCSharp.

We describe below how to test the first one, ContractModel.

Description of the model

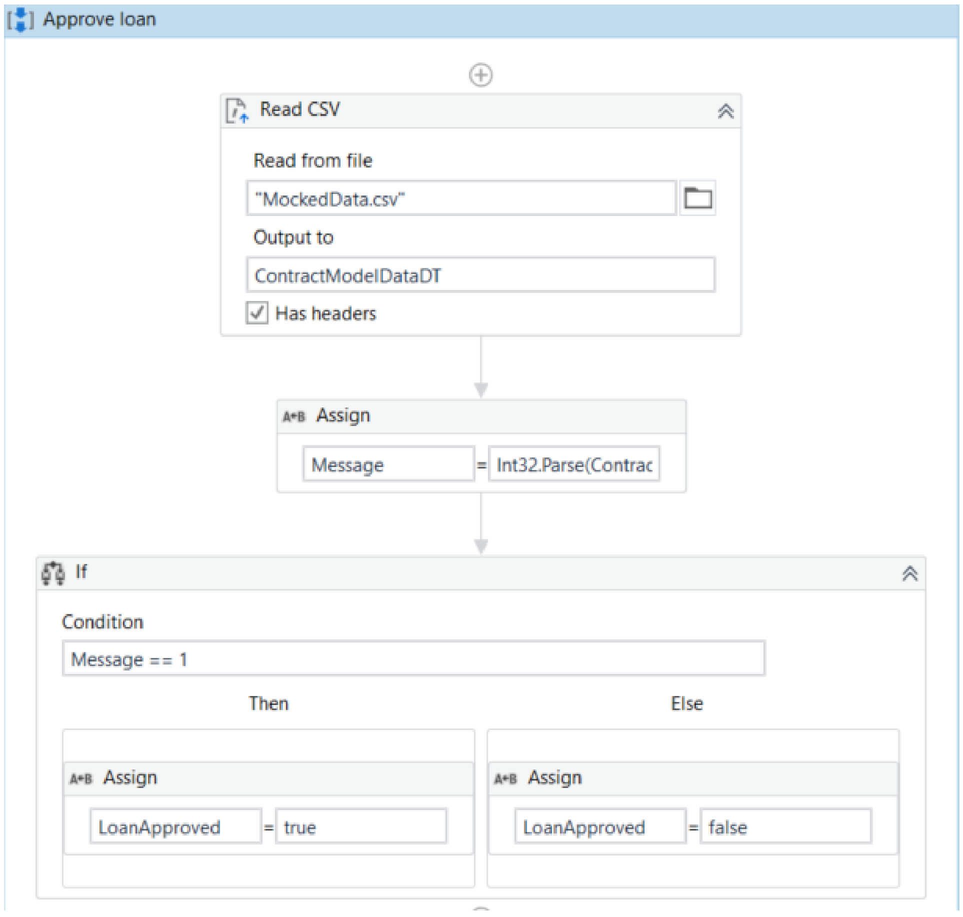


```
My Process
@Variables
{ "Amount":{"userInput":"True"}, "Message":{"userInput":"True","min":0,"max":1}, "ContractModelDataDT":{"path":"MockedData.csv"}}
```

The process is a toy example that automates the approval or denial of loans using UiBank Loan web application developed by UiPath. After the robot enters the loan data (the amount requested as a loan) and customer data (not relevant in the scenario) into the app, it creates a loan and verifies if the amount is larger than 20,000. If so, the robot executes the Approve loan sequence (business explanation: when the requested loan is large, an extra approval step is added), getting input from the user interface via a "Get text" activity in order to retrieve the message displayed in the application. The message can have two possible values – approved and not approved. If the loan is rejected (I.e., it is not approved), the process ends. If it is approved, the robot continues the process with Process Loan and End sequences. If the order amount is smaller than 20,000, the robot follows the path Process Loan -> End. Now let us focus on the conditions inside this model.

First, the condition inside the "Large Amount" Flow Decision, Amount >= 20,000, contains the variable Amount, which influences the robot to follow a specific path through the workflow. Therefore, our tool (the Fuzzer component) should be able to generate values for the Amount variable. We provide this information to the tool using an annotation in the main sequence of the flow. We explain the use of annotations in our testing tool in section 2.4.

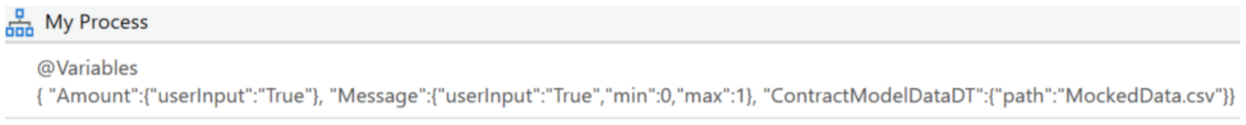
Second, in the Approve loan sequence, the robot gets a message input from the user interface using a "Get text" activity in order to retrieve the message displayed in the application. Since our tool does not control the UI input, we choose to mock this action, by using a .csv file "MockedData.csv", where the value of the Message variable will be found, instead of retrieving it from the user interface. This way, we can take advantage of the test data generated by the testing tool, copying the value of the Message variable in the ContractModel.csv for each test case, simulating this way the interaction with the user interface. To simplify a bit the explanations, we chose to assign to the Message variables the values 1 (for "Approved") or 0 (for "Not approved"), information needed also in the Annotation area.



Based on the values of the Message variable, the loan is approved or not.

Note that the condition inside the "Loan rejected?" Flow Decision, LoanApproved == true, is dependent on the Amount and Message variables (Amount < 20,000, Message == 1) so the path to follow is determined by the logic of the model.

The annotation required in order to generate the test cases for this flow is the following:

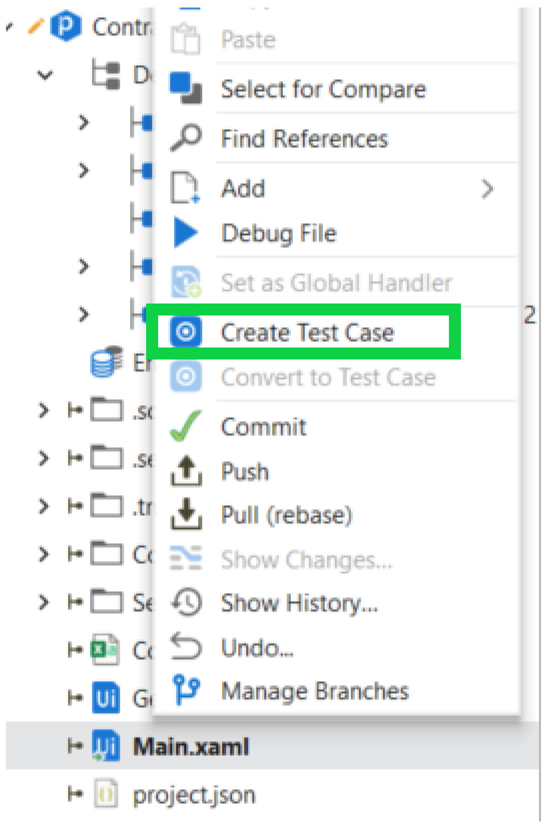


My Process

```
@Variables
{ "Amount":{"userInput":"True"}, "Message":{"userInput":"True","min":0,"max":1}, "ContractModelDataDT":{"path":"MockedData.csv"}}
```

Testing the ContractModel

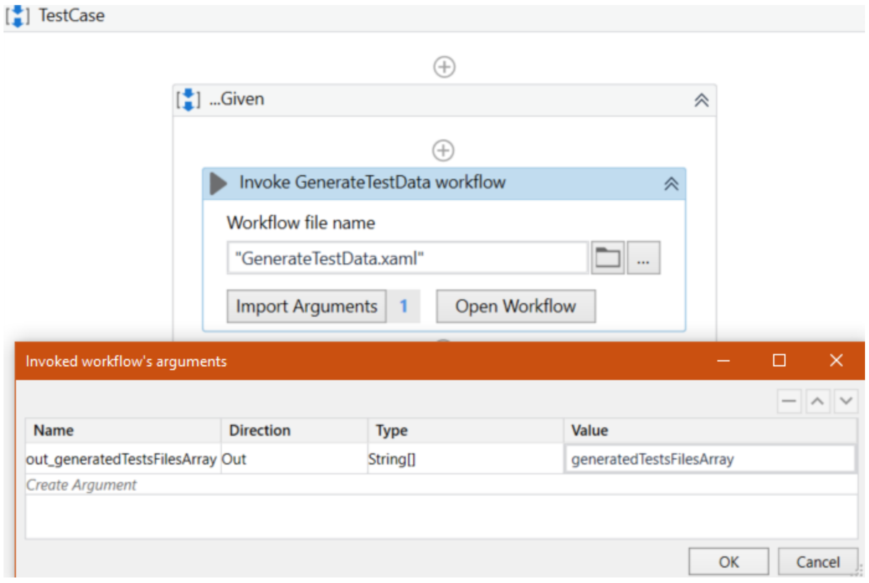
a. Open the model in UiPath Studio and create a new test case by right-clicking on the .xaml file you would like to test, in our case "Main.xaml". Choose *Create Test Case*.



A new .xaml file will be created with the predefined structure “Given”, “When”, “Then”.

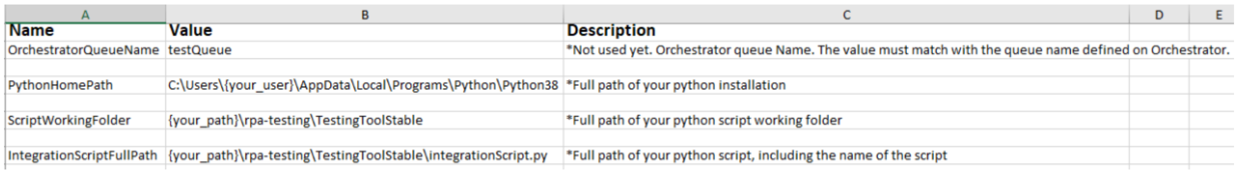
Copy the .xaml file GenerateTestData.xaml found in “Applications\ReusableTestingArtefacts” and paste it into your model folder.

In the “Given” section, invoke the newly added .xaml file (GeneratedTestData.xaml). This workflow has an output parameter representing the array of .csv files generated by the Fuzzer, each file containing test cases.



Name	Direction	Type	Value
out_generatedTestsFilesArray	Out	String[]	generatedTestsFilesArray

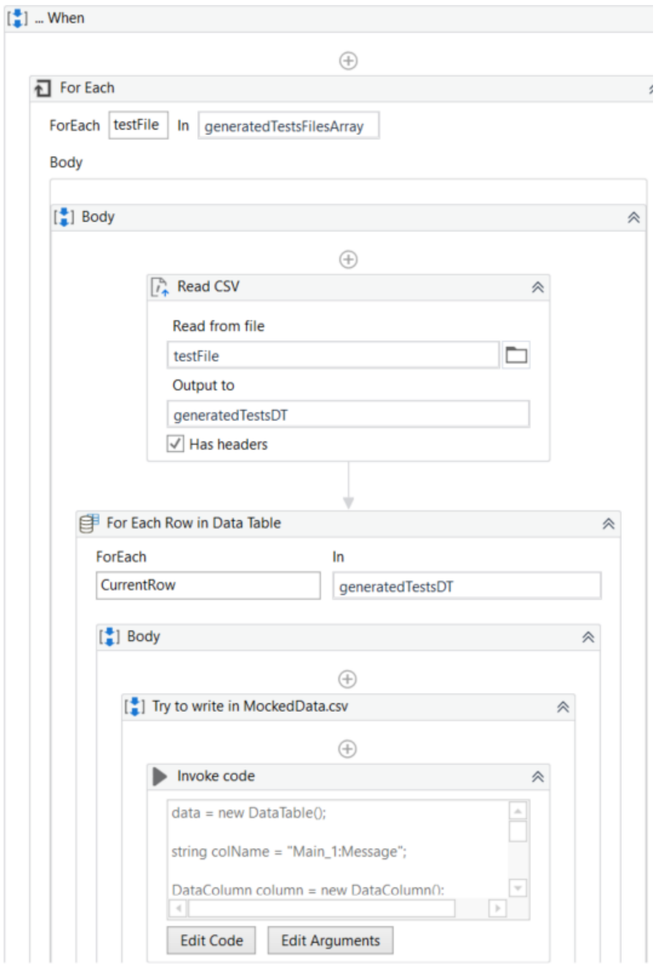
b. Copy the ConfigTestScript.xlsx found in "Applications\ReusableTestingArtefacts" and paste it into your model folder. This file contains the following parameters and it's used to configure the execution of the python script:



Name	Value	Description
OrchestratorQueueName	testQueue	*Not used yet. Orchestrator queue Name. The value must match with the queue name defined on Orchestrator.
PythonHomePath	C:\Users\{your_user}\AppData\Local\Programs\Python\Python38	*Full path of your python installation
ScriptWorkingFolder	{your_path}\rpa-testing\TestingToolStable	*Full path of your python script working folder
IntegrationScriptFullPath	{your_path}\rpa-testing\TestingToolStable\integrationScript.py	*Full path of your python script, including the name of the script

These parameters should be configured based on your requirements and following each parameter description found in the column "Description".

c. In the "When" section of your test case, you can execute the test cases from each generated test file.



... When

For Each

ForEach testFile In generatedTestsFilesArray

Body

Read CSV

Read from file: testFile

Output to: generatedTestsDT

☑ Has headers

For Each Row in Data Table

ForEach: CurrentRow In: generatedTestsDT

Body

Try to write in MockedData.csv

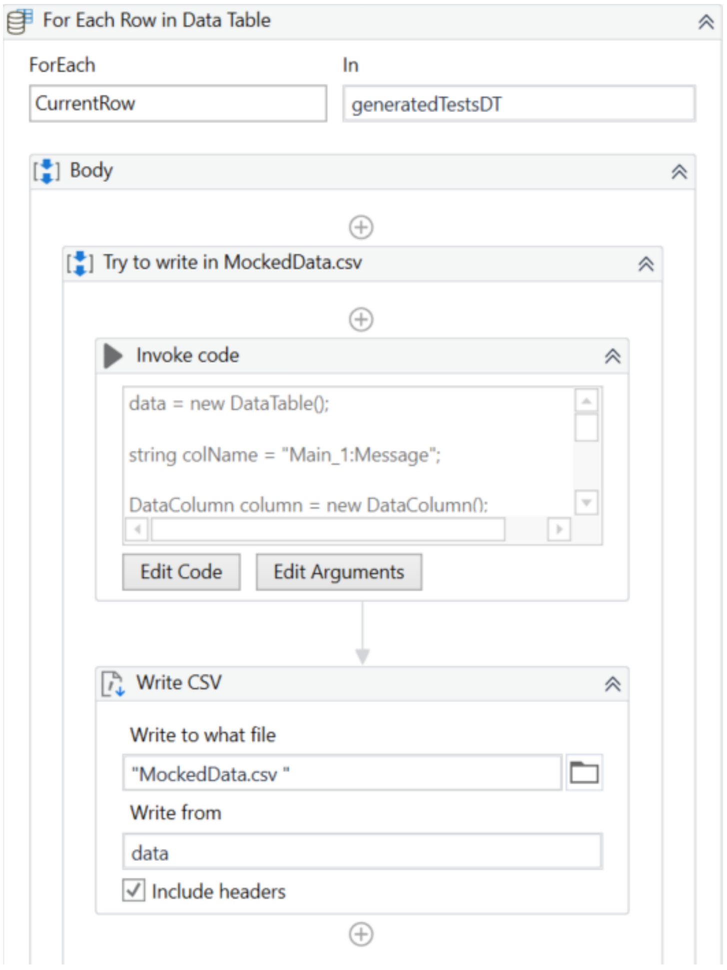
Invoke code

```
data = new DataTable();

string colName = "Main_1:Message";

DataColumn column = new DataColumn();
```

The output of the GenerateTestData.xaml invoked in the Given section is an array of strings representing each .csv containing the test cases, in our case let us call it generatedTestsFilesArray. For each test file from the generatedTestsFilesArray, we read the test file and, since we are using MockedData.csv in order to simulate the UI input for Message variable, we need to get from the generatedTests.csv file (generated by the Fuzzer) the test data for this variable.

```
data = new DataTable();

string colName = "Main_1:Message";

DataColumn column = new DataColumn();

column.DataType = System.Type.GetType("System.String");

column.ColumnName = colName;

data.Columns.Add(column);


var val = CurrentRow[colName].ToString();

DataRow row = data.NewRow();

row[colName] = val;

data.Rows.Add(row);
```

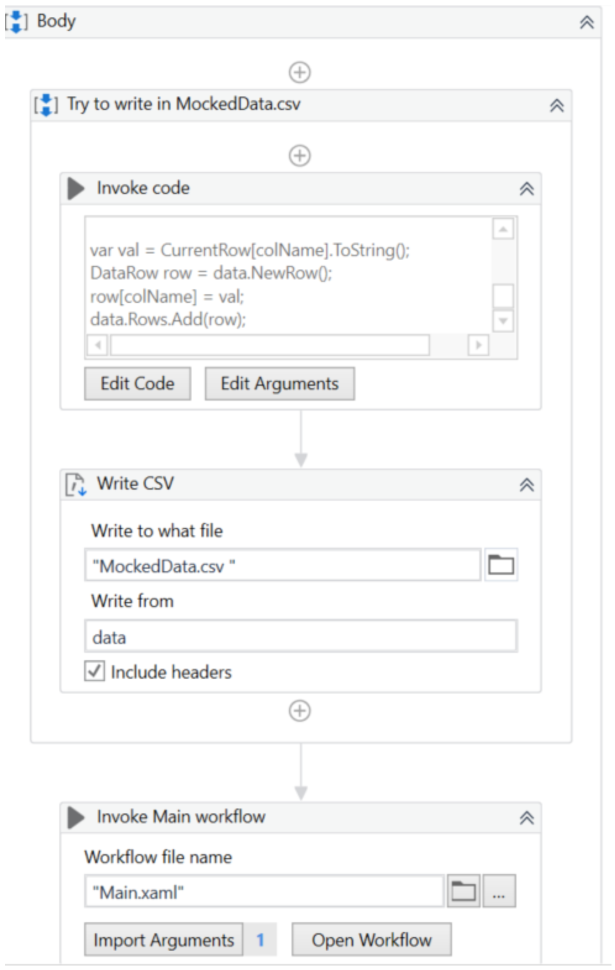
d. Afterwards, the test data for the Message variable must be written in the MockedData.csv, for each generated test case.


For Each Row in Data Table
ForEach: CurrentRow
In: generatedTestsDT
Body
Try to write in MockedData.csv
Invoke code

```
data = new DataTable();

string colName = "Main_1:Message";

DataColumn column = new DataColumn();
```

Write CSV
Write to what file: "MockedData.csv "
Write from: data
☑ Include headers

e. The last step will be to invoke the workflow under test:



```
var val = CurrentRow[colName].ToString();
DataRow row = data.NewRow();
row[colName] = val;
data.Rows.Add(row);
```

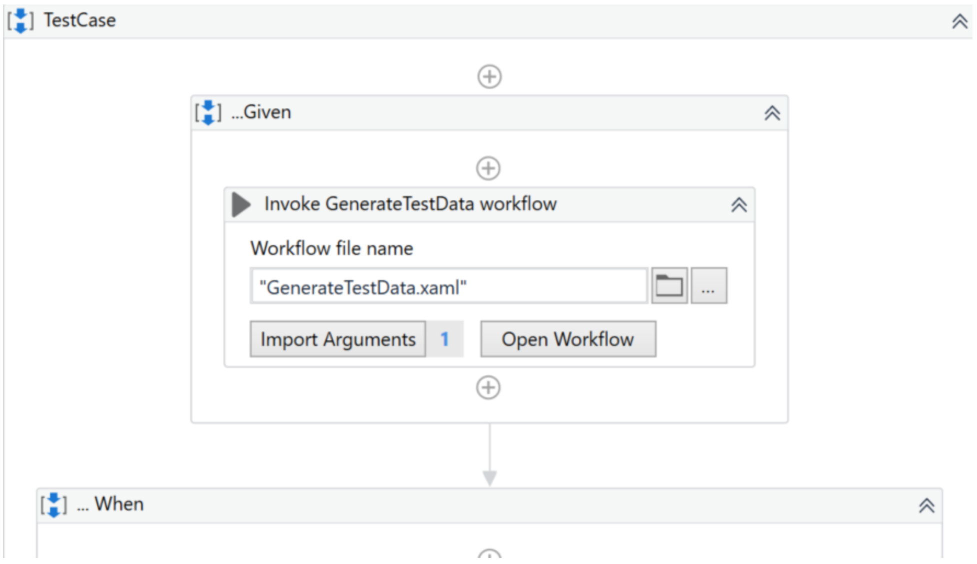
Write to what file: "MockedData.csv "

Write from: data

Workflow file name: "Main.xaml"

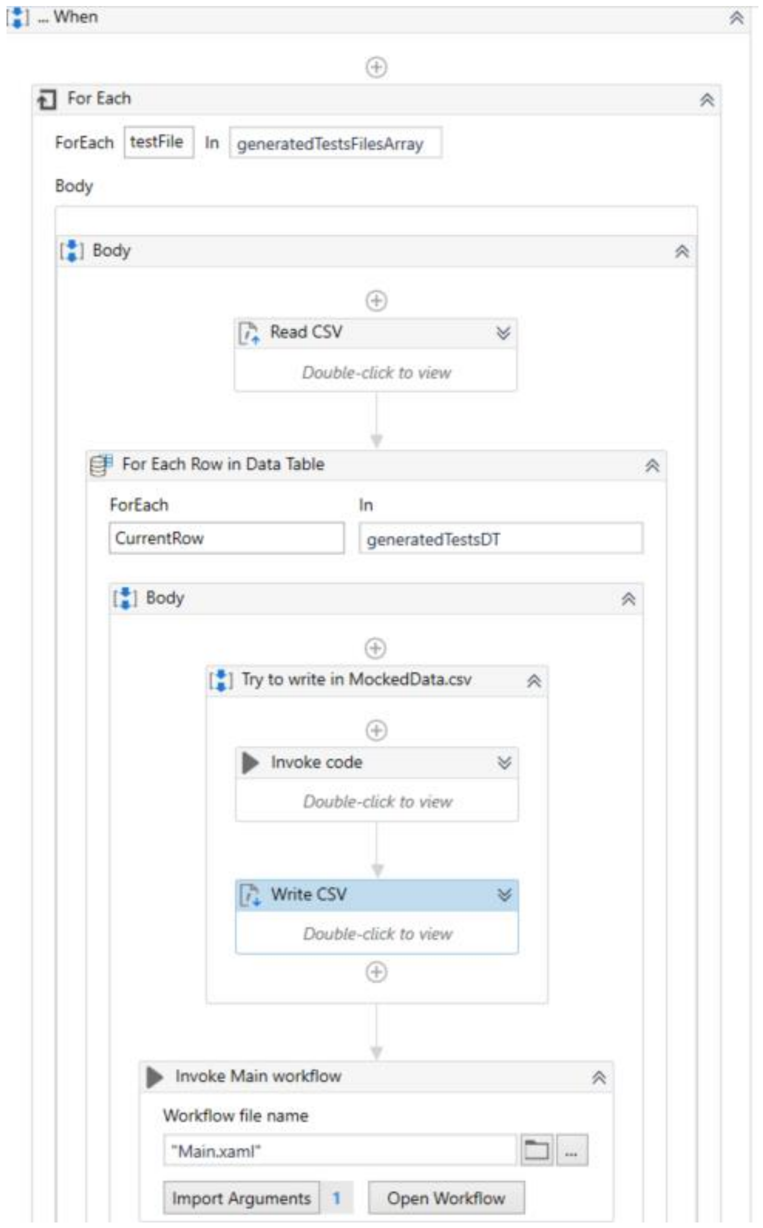
Import Arguments: 1

having as input argument the Amount variable generated in generatedTests.csv file.
The Test Case should look like this:

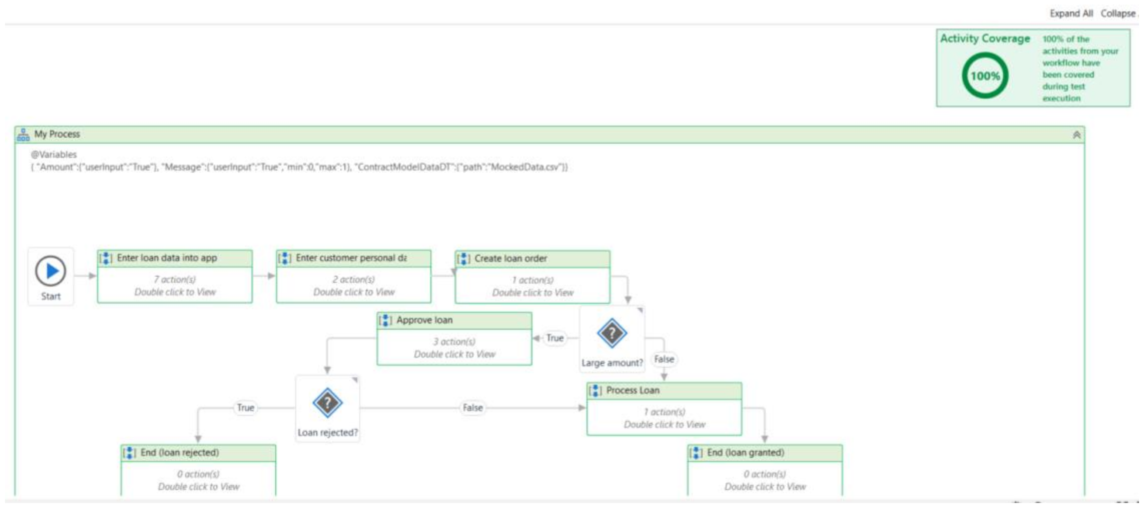


Workflow file name: "GenerateTestData.xaml"

Import Arguments: 1

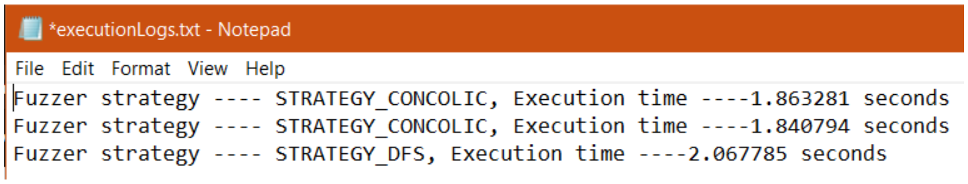


The activity coverage will be displayed in UiPath Studio:



```
@Variables
{ "Amount":{"userInput":"True"}, "Message":{"userInput":"True","min":0,"max":1}, "ContractModelDataDT":{"path":"MockedData.csv"}}
```

The generated test files can be found in the folder of the model. Also, the tool generates a log file containing details about the execution strategy and run time:



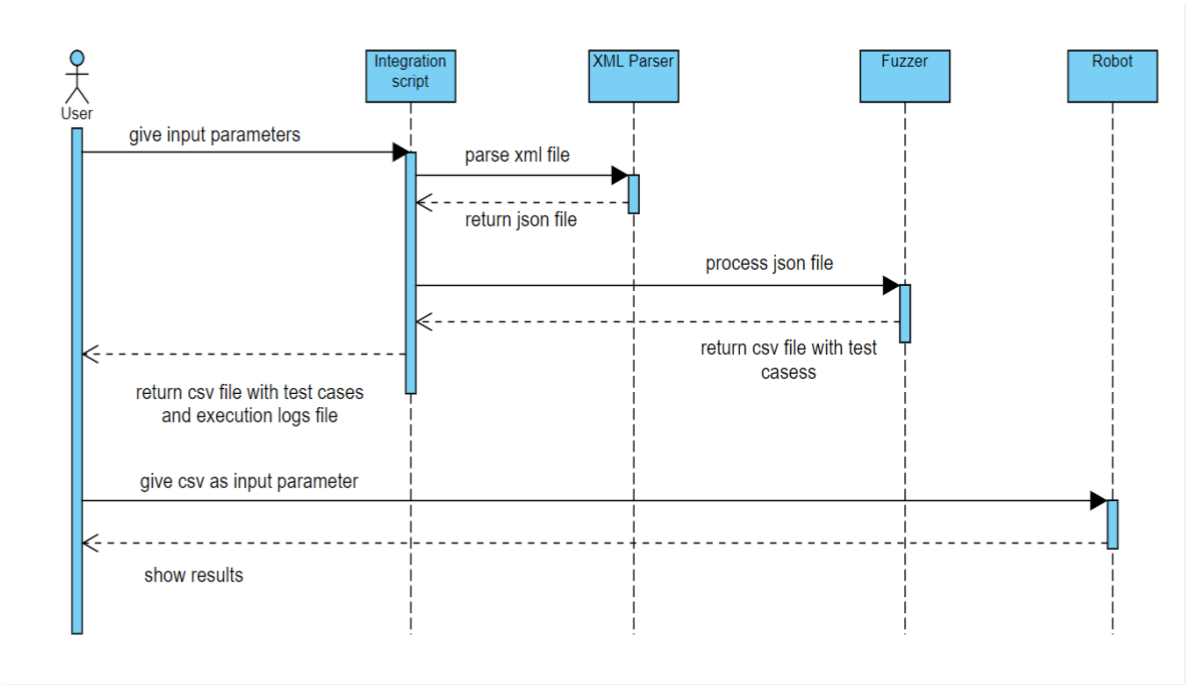
```
Fuzzer strategy ---- STRATEGY_CONCOLIC, Execution time ----1.863281 seconds
Fuzzer strategy ---- STRATEGY_CONCOLIC, Execution time ----1.840794 seconds
Fuzzer strategy ---- STRATEGY_DFS, Execution time ----2.067785 seconds
```

Note: Make sure the userInput.json file contains the specific parameters of the model. In the case of ContractModel, the userInput.json file should look like this (change the parameters based on your configuration):

```
{
  "xamlPath": "{PathToRpaTesting}\\rpa-
testing\\TestingToolStable\\Applications\\C#Models\\ContractModel\\Main.xaml",
  "modelBasePath": "{PathToRpaTesting} \\rpa-
testing\\TestingToolStable\\Applications\\C#Models\\ContractModel\\",
  "solverStrategy": "STRATEGY_DFS",
  "outputTests_MaxTestPerFile": 5,
  "debug_tests_fullPaths": 0,
  "debug_ConsoleOutput": 0,
  "debug_tests_fullVariablesContent": 0,
  "seedsFile": null,
  "numRandomGeneratedSeeds": 0
}
```

userInput.json can be found in the TestingToolStable directory.

1.2 Technical aspects



The process above presents the various components of the testing tool and their relations during the execution.

The implemented components are:

1. XMLParser aka Transformer from the xaml robot to an annotated graph format (suitable for the testing tool
2. TestGenerator aka Fuzzer
3. integrationScript.py - which connects the above components:
   A. XMLParser
   B. TestGenerator/Fuzzer

integrationScript.py gets as input the values from userInput.json (used to set up the XMLParser and the Fuzzer and must be filled in by the user at the beginning of the testing process, as a prerequisite). The purpose of the integrationScript.py, as its name states, it is to connect the two components, the XMLParser and Fuzzer: the XMLParser is used to parse .xaml files, the result being serialized in a .json file, whereas the Fuzzer accepts as input the .json file generated by the XMLParser. The execution of both components is performed with the help of the integrationScript.py, having as result the following files, stored in the model folder:

- outputXamlParser.json, representing the output of the XMLParser
- generatedTests_x.csv, where x >=0, representing the output of the Fuzzer
- executionLogs.csv, with the purpose of giving details about the execution of the Fuzzer, such as the solver strategy used and the execution time.
- debugGraph.png, representing the flow graph of the model.

In addition to these components, we have created a reusable workflow ("TestingToolStable\ReusableTestingArtefacts\GeneratedTestData.xaml") designed to execute the integrationScript.py and return as result an array of test files generated by the Fuzzer. In addition, ConfigTestScript.xlsx (found at "TestingToolStable\ReusableTestingArtefacts\Configuration") helps the user to set up the environment for running the integrationScript, providing the possibility to easily configure the parameters required to invoke a python script in the UiPath workflow.

Once the test sets are generated, the only remaining thing is to provide the file as an input for your test case.

2. Transforming a UiPath Workflow to Json using XMLParser tool

XMLParser is an application designed to parse UiPath workflows (".xaml" files) to a structure which is more suitable for test data generation (closer to a control flow graph CFG and easier to understand by an independent consumer). The result can be serialized in a ".json" file and contains data on all the reachable workflows (it supports InvokeWorkflow activities). Among the exported data, there are: variable and arguments names for every workflow, a graph composed by all reachable activities and additional information on some of them, etc.

2.1 How it works

XMLParser is built in an executable and can be run with parameters from the command line. To view the supported command and their description, one can run "XMLParsing.exe /help". The supported commands at the moment of writing this file are: "z3ConditionalGraph <workflow-file-path>", "z3FullGraph <workflow-file-path>", "z3ReducedGraph <workflow-file-path>". All of them receive the workflow file by its path. The full-graph command exports the whole graph composed of the parsed activities. The "reduced" one applies a "Reducer" over the graph and exports the most relevant activities and the "conditional" one exports only the conditional nodes (this command was used in the early development stages of the whole system and it is not stable anymore now). The most used command is the one that reduces the workflow since it only exports the one the consumer might be interested in.

2.2. How is the application designed?

The parser code is structured in three main file categories: Common, Utils and Services. The parsing process involves parsing the activities in one or more nodes to reach the following result: a graph with two types of branches, "true" and "false" branches. When a node in the graph does not have an expression, then it is considered that the only way to go is the "true" branch and, when it has one, then the execution thread is conditioned by that expression. Therefore, the Common folder contains the models the app uses to represent the graph: graph, nodes, transitions, and a workflow data class. There are also node extensions that may contain more data on certain activities (for example, assign activities contain To and Value fields for left and right operands). The Utils folder contains some utility classes, one for parsing Annotations, one for parsing Expressions (based on a Chain-of-Responsibility pattern), one for Reflection, etc. The process of getting from a ".xaml" file to a ".json" file involves a pipeline of processes which are handled by three types of classes in the code: Parsers, Reducers and Serializers. A Parser class is a class that knows how to parse an activity based on its type. There is a convention that when parsing an activity, a Parser must return a start and an end node for the output structure to be integrated easily in the final structure. The Parsers are instantiated based on a Factory pattern. The Reducers receive the output graph and reduce the nodes based on their importance. The Serializers can serialize the parsed structure in a file (or in the standard output).

2.3. How can one easily add support for a new activity?

All the activities are supported, but in a straightforward way: the default parser creates a Node for that activity, and it only considers its name and its transitions to other Activities (not inner activities contained by it). Based on the information presented in the previous chapter, to add custom support for a new activity, one should create a new instance of Parser by extending the *IActivityParser* interface and registering it in the *ActivityParserFactory* class. Also, if additional data must be kept on the Node, then one should also create a derived class and place it in the *NodeExtensions* folder. The process may be more difficult in case of more complex activities (TryCatch, for example).

2.4 Annotation support. What is it?

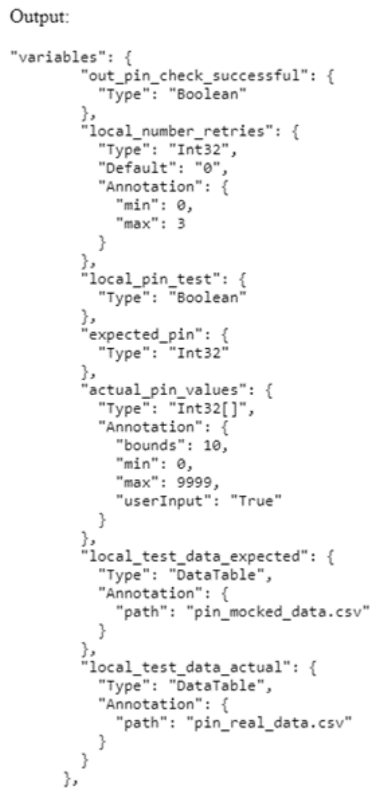
Annotations represent a direct way of communicating data from a workflow file to the consumer of the ".json" file created by the parser. The business reason may be because of the incompatibilities of the expression written in C# with the language the consumer is written in or because the consumer needs more data on the workflow to provide reliable results (on the data type of the variables, their bounds, patterns, etc.). The annotation supported provided by the XMLParser is relying on the Annotation (commentaries) support implemented in the UiPath Studio.

There are two types annotations supported: *@Variables* and *@Expression*. The first one is used to provide more data on the variables and arguments from the workflow. An annotation looks like this:

@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"}}

The current version requires this annotation to be placed at the top-level activity of a workflow, in order to be correctly interpreted. The keyword *@Variables* must be followed by a valid JSON object whose keys are valid names of the variables and arguments included in the workflow. The values must also be JSON objects that would be pasted in the output file as they are written in the annotation. An example of output for the above annotation is presented in the picture below.



Output:

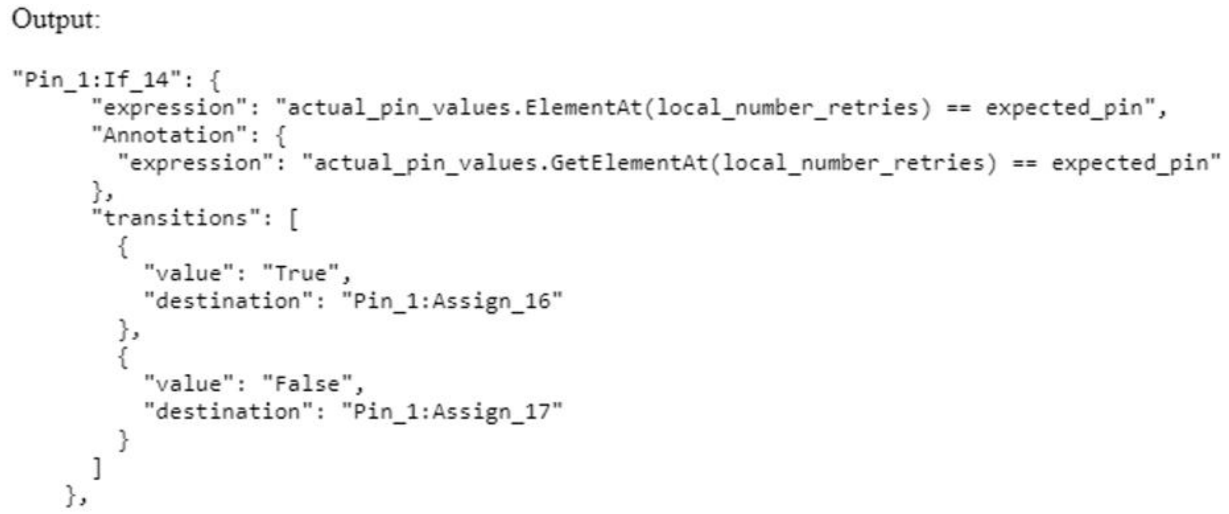
```
"variables": {
       "out_pin_check_successful": {
         "Type": "Boolean"
       },
       "local_number_retries": {
         "Type": "Int32",
         "Default": "0",
         "Annotation": {
           "min": 0,
           "max": 3
         }
       },
       "local_pin_test": {
         "Type": "Boolean"
       },
       "expected_pin": {
         "Type": "Int32"
       },
       "actual_pin_values": {
         "Type": "Int32[]",
         "Annotation": {
           "bounds": 10,
           "min": 0,
           "max": 9999,
           "userInput": "True"
         }
       },
       "local_test_data_expected": {
         "Type": "DataTable",
         "Annotation": {
           "path": "pin_mocked_data.csv"
         }
       },
       "local_test_data_actual": {
         "Type": "DataTable",
         "Annotation": {
           "path": "pin_real_data.csv"
         }
       }
     },
```

The second type of supported annotation is *@Expression.* Its purpose is to provide an alternative expression, compatible with the consumer or in a more friendly format. It should be placed on the activity whose expression is to be simplified and it should also resemble a valid JSON object. The object should have a key with the name “expression” and the value should be a string that represents the simplified version of the expression. It may look like this:

@Expression

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

The result would look like in the following picture:



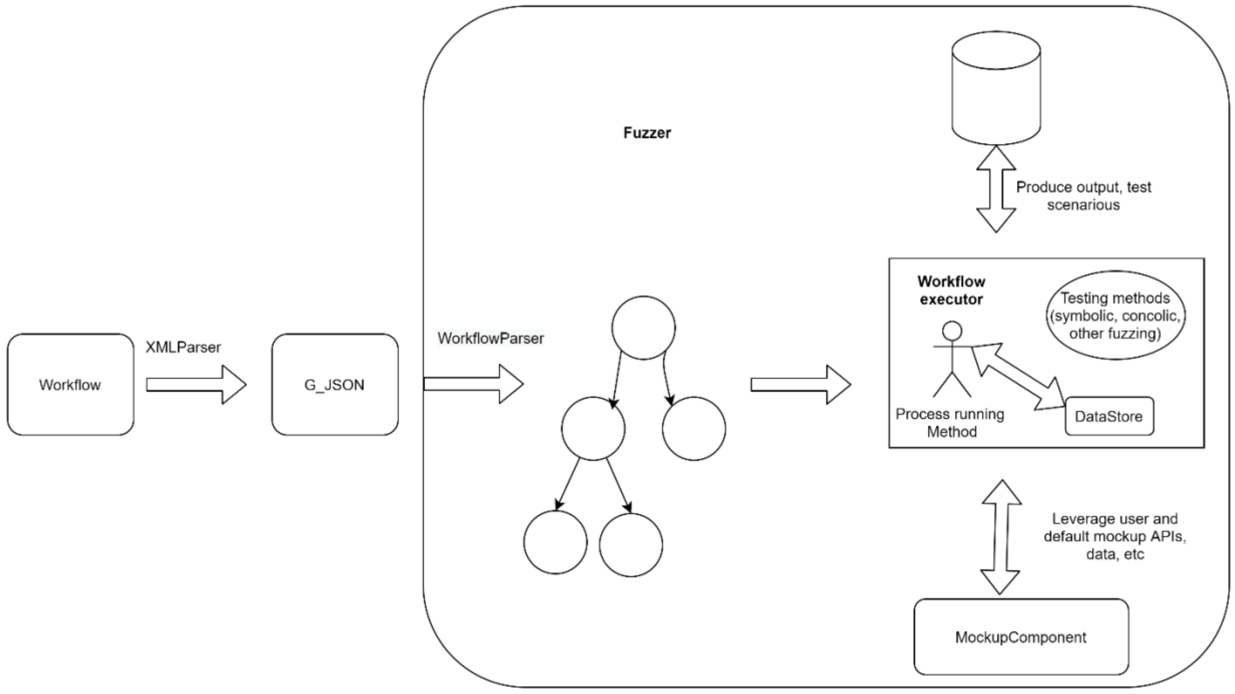
Output:

```
"Pin_1:If_14": {
      "expression": "actual_pin_values.ElementAt(local_number_retries) == expected_pin",
      "Annotation": {
        "expression": "actual_pin_values.GetElementAt(local_number_retries) == expected_pin"
      },
      "transitions": [
        {
          "value": "True",
          "destination": "Pin_1:Assign_16"
        },
        {
          "value": "False",
          "destination": "Pin_1:Assign_17"
        }
      ]
    },
```

3. Technical foundations of the testing tool

In Section 1.2, we already provided an overview of the tool architecture and execution flow. In this section we provide in-depth technical details of the research aspects and implementation choices.

First, we transform the xaml-type workflow graph W into a JSON-type graph G_JSON. Then, we transform G_JSON to an internal Abstract Syntax Tree (AST) that transforms nodes and expressions into an understandable format for the Fuzzer process. The Fuzzer process is then used with different methods to produce as output a database of test scenarios.


Workflow
XMLParser
G_JSON
WorkflowParser
Fuzzer
Produce output, test scenarious
Workflow executor
Testing methods (symbolic, concolic, other fuzzing)
Process running Method
DataStore
Leverage user and default mockup APIs, data, etc
MockupComponent

3.1 The WorkflowParser implementation details and annotations

The annotations support detailed in the previous section (Section 2) are very important for the quality of the testing process. Instead of having the testing process spending significant resources on paths that are not feasible from a practical perspective at all, boundaries and patterns are defined inside annotations. Thus, using the human in the loop, one can give important hints to the min/max of individual variables, content of arrays or data tables entries, boundaries of arrays or sequence of numbers, patterns for strings, etc.

As already mentioned, the JSON graph G_JSON is transformed into an Abstract Syntax Tree (AST). The source code is located in class WorkflowParser. A graph structure containing nodes and parsed expressions into an understandable format for the test generation tool is formed at this point in an object instance called WorkflowGraph. The WorkflowGraph holds the structure of the AST transformed workflow. Each node is a ASTFuzzerNode. Multiple types of nodes like comparators, logical and math operators are defined and supported in an extensible system up to the user need (see the code in Parser_ASTNodes.py for an exhaustive list of them).

3.2 The WorkflowExecutor implementation details

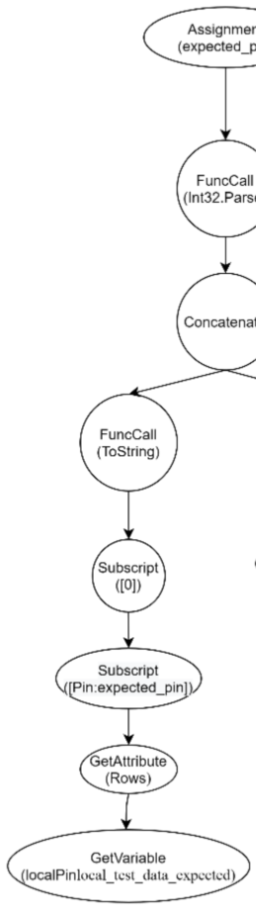
A DataStore componet holds the concrete values of the variables and symbolic instances of variables that are tainted by the user’s input. SymbolicVars (D in DataStore) is the list of symbolic variables that the testing process iterates on. In the annotations side we have the “userInput” specification to define which variables are considered as input to the user and become directly symbolic variables. Any other variable that is connected to them (tainting) becomes symbolic as well in the execution flow. There is also a member that defines the current concrete values of each variable, both symbolic or non-symbolic: Values(D).

The WorkflowExecutor (implemented in ASTFuzzerNodeExecutor) can execute an ASTFuzzerNode type, corresponding to its type (check function executeNode of class ASTFuzzerNodeExecutor). Executing a node means evaluating the expression defined in the Workflow for that particular node. Each expression Expr in Node can produce side effects visible in the contextual DataStore instance D that the WorkflowExecutor is working on. It is important to understand that the evaluation of the expression should be done in real time. An example is given below.

Let us take one of the expressions in the BankLoanWithPin model:

> *expected_pin = Int32.Parse(local_test_data_expected.Rows[0][\"Pin:expected_pin\"].ToString()) + localPinSuffix.ToString())*

After WorkflowParser parses this expression, it is transformed into an AST as below:

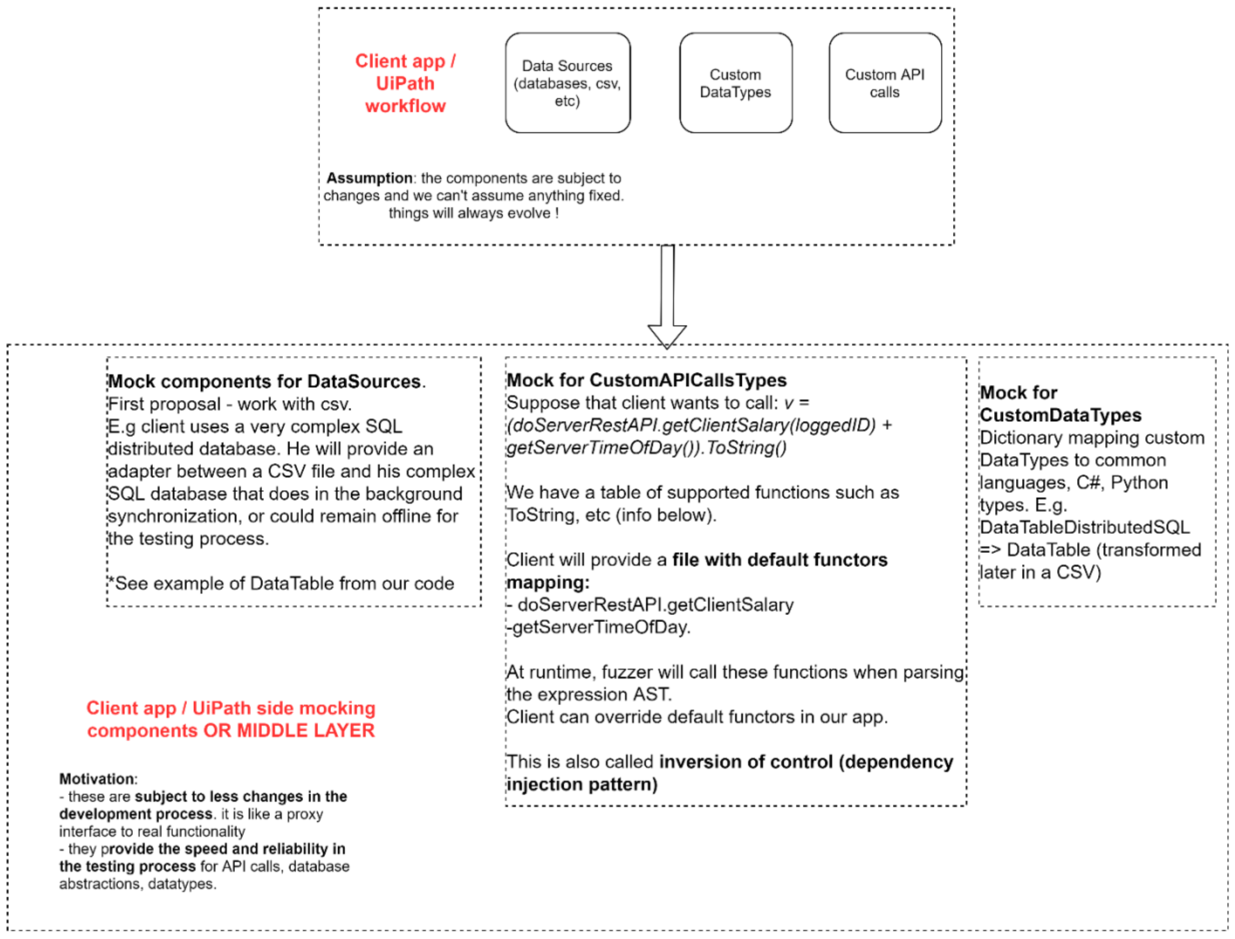

Assignment (expected_pin)
FuncCall (Int32.Parse)
Concatenate
FuncCall (ToString)
FuncCall (ToString)
Subscript ([0])
GetVariable (localPinSuffix)
Subscript ([Pin:expected_pin])
GetAttribute (Rows)
GetVariable (localPinlocal_test_data_expected)

In the AST leaves there are the *GetVariable* nodes that retrieve that data out from the *DataStore*, then some processing is done to the expression up to the top of the tree. It is important to note that the data table being invoked (*localPinlocal_test_data_expected* being a *DataType* type), variables *localPinSuffix* and *expected_pin* index needed to query the data table can be modified during the execution process. Thus, the only valid way is to evaluate these expressions at runtime. After running this expression with its side effects, the *DataStore* instance is updated with the new value for variable *expected_pin*, *Values(D) = Values(D) + {expected_pin = Eval(node.expression)}*.

The WorkflowExecutor executes the graph, being coordinated by the testing methods defined in the next sections.

3.3 Mocking strategy

The idea behind mocking is to allow Clients to inject their own mocking types for both data sources and code. The overall idea is defined in the figure below:


Client app / UiPath workflow
Data Sources (databases, csv, etc)
Custom DataTypes
Custom API calls
Assumption: the components are subject to changes and we can't assume anything fixed. things will always evolve !
Mock components for DataSources.
First proposal - work with csv.
E.g client uses a very complex SQL distributed database. He will provide an adapter between a CSV file and his complex SQL database that does in the background synchronization, or could remain offline for the testing process.
*See example of DataTable from our code
Mock for CustomAPICallsTypes
Suppose that client wants to call: *v = (doServerRestAPI.getClientSalary(loggedID) + getServerTimeOfDay()).ToString()*
We have a table of supported functions such as ToString, etc (info below).
Client will provide a **file with default functors mapping:**
- doServerRestAPI.getClientSalary
-getServerTimeOfDay.
At runtime, fuzzer will call these functions when parsing the expression AST.
Client can override default functors in our app.
This is also called **inversion of control (dependency injection pattern)**
Mock for CustomDataTypes
Dictionary mapping custom DataTypes to common languages, C#, Python types. E.g. DataTableDistributedSQL => DataTable (transformed later in a CSV)
Client app / UiPath side mocking components OR MIDDLE LAYER
Motivation:
- these are **subject to less changes in the development process**. it is like a proxy interface to real functionality
- they **provide the speed and reliability in the testing process** for API calls, database abstractions, datatypes.

Examples in the source code can be found in the following files

- **Datatypes APIs and data sources mocking**, *Parser_DataTypes.py*: The purpose of this file is to offer a default implementation of data types. The user can then import a file or dictionary or inject through the sys modules API new Data Types! By default, we support a few *DataTypes* common in C# and a kind of *DataTables* to create mockups for databases through csv files.
- **Global functions APIs moking**, *Parser_Functions.py*: The purpose of this file is to offer a default implementation of certain global functions. The client can then import its own function set here to extend the system.

3.4 Results streaming

Some explanations on how can the results are displayed.

3.4.1 Base output systems

The results will be displayed using these options:

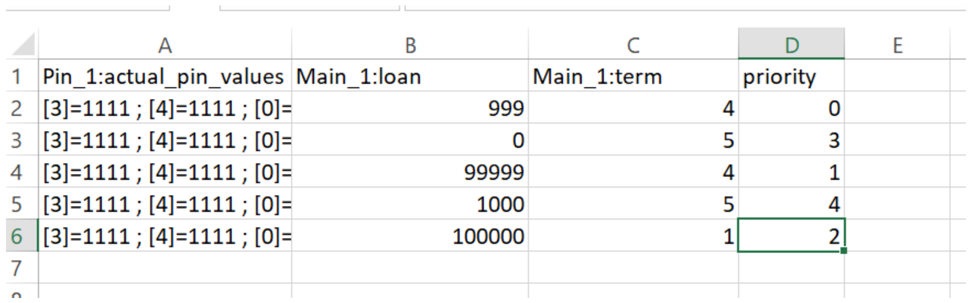
-**outputTests**_PrefixFile

"generatedTests" - how to prefix the output streams csv with tests

-outputTests_MaxTestPerFile

5 - an integer value

The parameter ***outputTests_PrefixFile*** is a path and prefix to where to write and name the csv output files, with a row representing a test case, and a column representing the variables content, as depicted in the sceeenshot below. The header contains the description of the variable name.



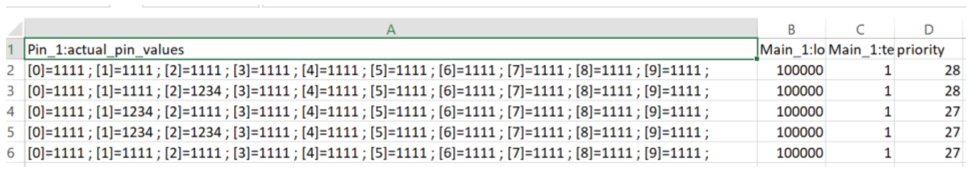
	A	B	C	D	E
1	Pin_1:actual_pin_values	Main_1:loan	Main_1:term	priority	
2	[3]=1111 ; [4]=1111 ; [0]=	999	4	0	
3	[3]=1111 ; [4]=1111 ; [0]=	0	5	3	
4	[3]=1111 ; [4]=1111 ; [0]=	99999	4	1	
5	[3]=1111 ; [4]=1111 ; [0]=	1000	5	4	
6	[3]=1111 ; [4]=1111 ; [0]=	100000	1	2	
7					

See section about *Debugging support* for more options for the output.

The parameter ***outputTests_MaxTestPerFile*** defines how many test cases do you want to be produced on each csv output stream. If you set this to 5 and there will be 100 test cases in total, our method produces 20 output csv streams.. The purpose is to avoid read/write concurrency and to avoid blocking the test cases producing for longer period of times, i.e., yielding more parallelism between a producer - the Fuzzer method, and a consumer - an entity who wants to ingest the produced tests.

3.4.2 How arrays are displayed

Arrays are displayed in the format "[index] = value". This is because Fuzzer may access / need to use only parts of a broad array, not all indices! See the image below:



	A	B	C	D
1	Pin_1:actual_pin_values	Main_1:lo	Main_1:te	priority
2	[0]=1111 ; [1]=1111 ; [2]=1111 ; [3]=1111 ; [4]=1111 ; [5]=1111 ; [6]=1111 ; [7]=1111 ; [8]=1111 ; [9]=1111 ;	100000	1	28
3	[0]=1111 ; [1]=1111 ; [2]=1234 ; [3]=1111 ; [4]=1111 ; [5]=1111 ; [6]=1111 ; [7]=1111 ; [8]=1111 ; [9]=1111 ;	100000	1	28
4	[0]=1111 ; [1]=1234 ; [2]=1111 ; [3]=1111 ; [4]=1111 ; [5]=1111 ; [6]=1111 ; [7]=1111 ; [8]=1111 ; [9]=1111 ;	100000	1	27
5	[0]=1111 ; [1]=1234 ; [2]=1234 ; [3]=1111 ; [4]=1111 ; [5]=1111 ; [6]=1111 ; [7]=1111 ; [8]=1111 ; [9]=1111 ;	100000	1	27
6	[0]=1111 ; [1]=1111 ; [2]=1111 ; [3]=1111 ; [4]=1111 ; [5]=1111 ; [6]=1111 ; [7]=1111 ; [8]=1111 ; [9]=1111 ;	100000	1	27

3.4.3 Why some of the columns are missing?

Some columns values could be missing in the case that they are not touched or relevant for the given test. So, you must be prepared for EMPTY content of certain values. This means that you can put there a random value if you want.

3.4.4 Priorities tab:

The priority column, as show in the figures above is like a value of what our method thinks about the prioritization of that input. It can be customized and computed by the user through hooks (see the Methods and Strategies section). Someone who ingest the tests could ignore it or use it as given.

3.5 Debugging support and results displaying options

It is important to understand our supported debugging strategies and how to extend these capabilities. Currently we support the following debugging parameters:

-**debug_outputGraphFile**

"debugGraph.png" - a path local from the baseModelPath parameter, could be empty string if you do not need to see the internal graph structure.

-debug_tests_fullPaths

0 or 1

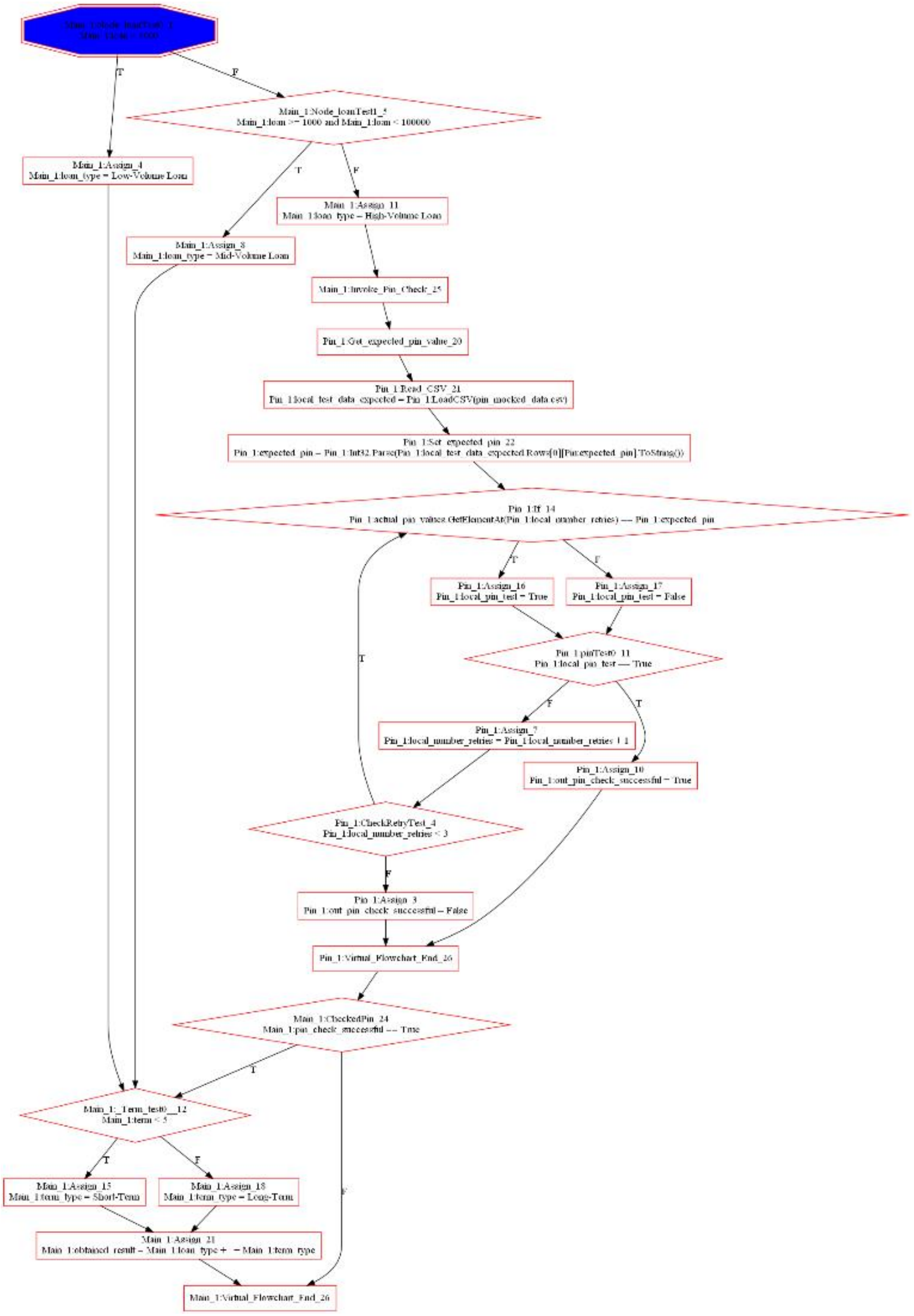
-debug_consoleOutput

0 or 1

-debug_tests_fullVariablesContent

0 or 1

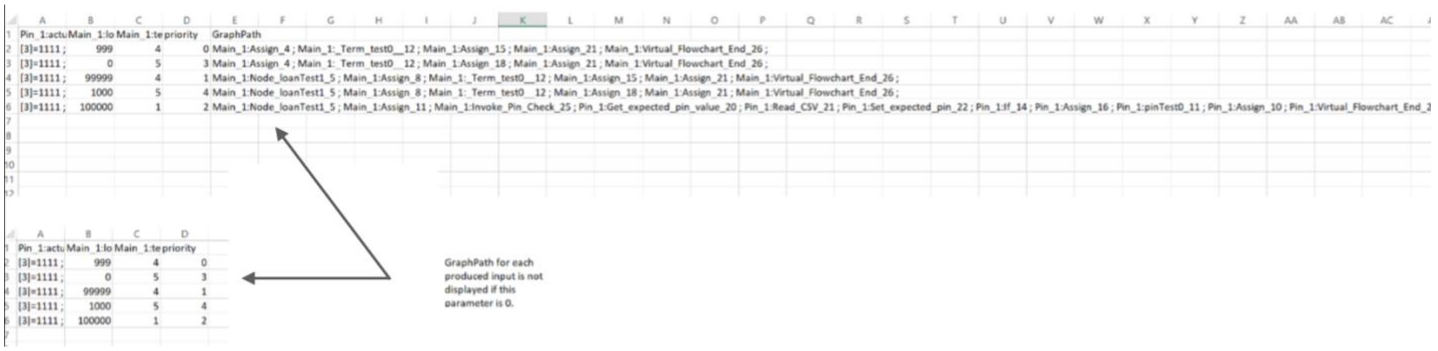
A. If **debug_outputGraphFile** is not an empty string you will see at the given path a graph output with the full SMT path transformed as defined in 3.A/3.B containing the graph with activities, conditions, branches, etc.


Main_1:Node_loanTest0_1
Main_1:loan < 1000
T
F
Main_1:Node_loanTest1_5
Main_1:loan >= 1000 and Main_1:loan < 100000
Main_1:Assign_4
Main_1:loan_type = Low-Volume Loan
T
F
Main_1:Assign_11
Main_1:loan_type = High-Volume Loan
Main_1:Assign_8
Main_1:loan_type = Mid-Volume Loan
Main_1:Invoke_Pin_Check_25
Pin_1:Get_expected_pin_value_20
Pin_1:Read_CSV_21
Pin_1:local_test_data_expected = Pin_1:LoadCSV(pin_mocked_data.csv)
Pin_1:Set_expected_pin_22
Pin_1:expected_pin = Pin_1:Int32.Parse(Pin_1:local_test_data_expected.Rows[0][Pin:expected_pin].ToString())
Pin_1:If_14
Pin_1:actual_pin_values.GetElementAt(Pin_1:local_number_retries) == Pin_1:expected_pin
T
F
Pin_1:Assign_16
Pin_1:local_pin_test = True
Pin_1:Assign_17
Pin_1:local_pin_test = False
T
Pin_1:pinTest0_11
Pin_1:local_pin_test == True
F
T
Pin_1:Assign_7
Pin_1:local_number_retries = Pin_1:local_number_retries + 1
Pin_1:Assign_10
Pin_1:out_pin_check_successful = True
Pin_1:CheckRetryTest_4
Pin_1:local_number_retries < 3
F
Pin_1:Assign_3
Pin_1:out_pin_check_successful = False
Pin_1:Virtual_Flowchart_End_26
Main_1:CheckedPin_24
Main_1:pin_check_successful == True
T
Main_1:_Term_test0__12
Main_1:term < 5
T
F
F
Main_1:Assign_15
Main_1:term_type = Short-Term
Main_1:Assign_18
Main_1:term_type = Long-Term
Main_1:Assign_21
Main_1:obtained_result = Main_1:loan_type + " " + Main_1:term_type
Main_1:Virtual_Flowchart_End_26

B. **debug_tests_fullPaths**

This is a costly debugging method so be sure you **do not activate it in production code.**

This will also make a difference between storing and output the full path that a new input would produce, or not. See an example below:



	A	B	C	D	E
1	Pin_1:actu	Main_1:lo	Main_1:te	priority	GraphPath
2	[3]=1111 ;	999	4	0	Main_1:Assign_4 ; Main_1:_Term_test0__12 ; Main_1:Assign_15 ; Main_1:Assign_21 ; Main_1:Virtual_Flowchart_End_26 ;
3	[3]=1111 ;	0	5	3	Main_1:Assign_4 ; Main_1:_Term_test0__12 ; Main_1:Assign_18 ; Main_1:Assign_21 ; Main_1:Virtual_Flowchart_End_26 ;
4	[3]=1111 ;	99999	4	1	Main_1:Node_loanTest1_5 ; Main_1:Assign_8 ; Main_1:_Term_test0__12 ; Main_1:Assign_15 ; Main_1:Assign_21 ; Main_1:Virtual_Flowchart_End_26 ;
5	[3]=1111 ;	1000	5	4	Main_1:Node_loanTest1_5 ; Main_1:Assign_8 ; Main_1:_Term_test0__12 ; Main_1:Assign_18 ; Main_1:Assign_21 ; Main_1:Virtual_Flowchart_End_26 ;
6	[3]=1111 ;	100000	1	2	Main_1:Node_loanTest1_5 ; Main_1:Assign_11 ; Main_1:Invoke_Pin_Check_25 ; Pin_1:Get_expected_pin_value_20 ; Pin_1:Read_CSV_21 ; Pin_1:Set_expected_pin_22 ; Pin_1:If_14 ; Pin_1:Assign_16 ; Pin_1:pinTest0_11 ; Pin_1:Assign_10 ; Pin_1:Virtual_Flowchart_End_2

	A	B	C	D
1	Pin_1:actu	Main_1:lo	Main_1:te	priority
2	[3]=1111 ;	999	4	0
3	[3]=1111 ;	0	5	3
4	[3]=1111 ;	99999	4	1
5	[3]=1111 ;	1000	5	4
6	[3]=1111 ;	100000	1	2

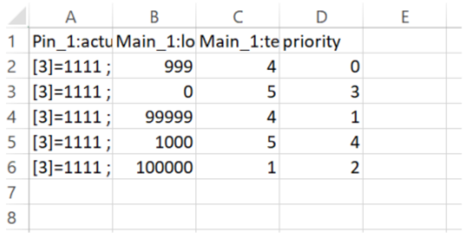
GraphPath for each produced input is not displayed if this parameter is 0.

C. **debug_consoleOutput**

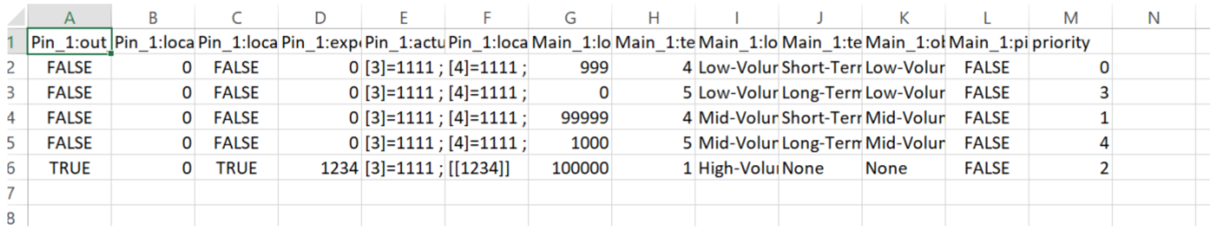
One can use this with 0 on production code in order to suppress the prints and everything. (One needs to let the environment console clean.) There is a commented code at top of the main.py script to find people and code who writes to stdout without checking this option.

D. debug_tests_fullVariablesContent

This option should not be activated in production code in general. Its role is to make the difference between displaying ALL the variables content at each test produced or displaying only the NEEDED variables that affect the model at all (I.e., those Annotated with *isUserInput = true*). You can think about it like this: The model produces a test case while driving through the graph. On this run it will touch many of the variables (possibly, not all). If this option is enabled, it will output all the variables in the model. If not, it will show only those important. Check the image below to see the output difference between activated/not activated:



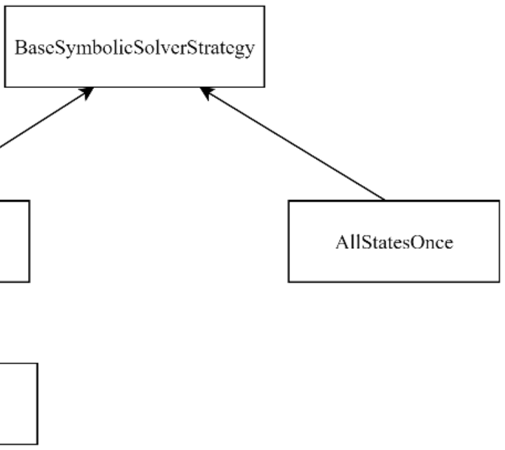
	A	B	C	D	E
1	Pin_1:actu	Main_1:lo	Main_1:te	priority	
2	[3]=1111 ;	999	4	0	
3	[3]=1111 ;	0	5	3	
4	[3]=1111 ;	99999	4	1	
5	[3]=1111 ;	1000	5	4	
6	[3]=1111 ;	100000	1	2	
7					
8					



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Pin_1:out	Pin_1:loca	Pin_1:loca	Pin_1:exp	Pin_1:actu	Pin_1:loca	Main_1:lo	Main_1:te	Main_1:lo	Main_1:te	Main_1:ol	Main_1:pi	priority	
2	FALSE	0	FALSE	0	[3]=1111 ;	[4]=1111 ;	999	4	Low-Volur	Short-Terr	Low-Volur	FALSE	0	
3	FALSE	0	FALSE	0	[3]=1111 ;	[4]=1111 ;	0	5	Low-Volur	Long-Term	Low-Volur	FALSE	3	
4	FALSE	0	FALSE	0	[3]=1111 ;	[4]=1111 ;	99999	4	Mid-Volur	Short-Terr	Mid-Volur	FALSE	1	
5	FALSE	0	FALSE	0	[3]=1111 ;	[4]=1111 ;	1000	5	Mid-Volur	Long-Term	Mid-Volur	FALSE	4	
6	TRUE	0	TRUE	1234	[3]=1111 ;	[[1234]]	100000	1	High-Volu	None	None	FALSE	2	
7														
8														

4. Strategies and methods for performing tests

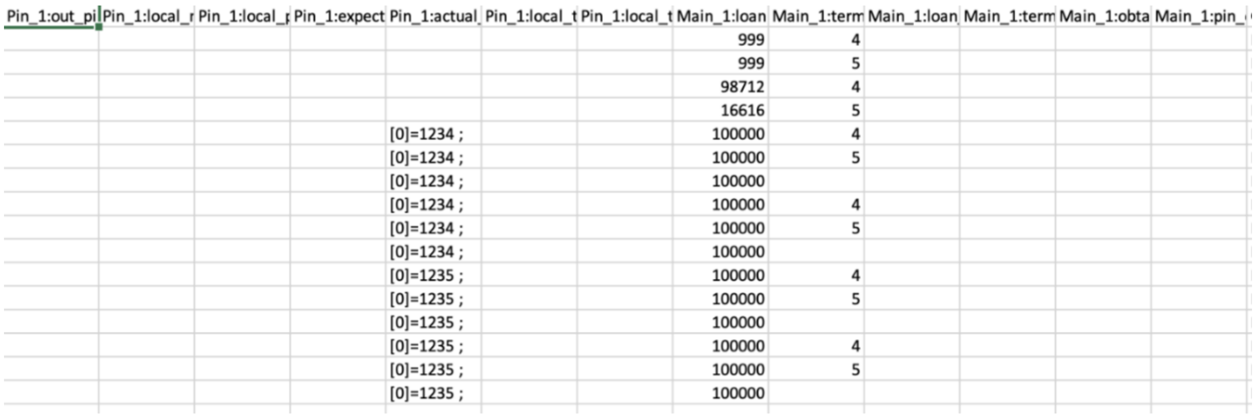
Currently there are three main strategies used in our implementation, all derived from a base class and explained below in details.



4.1 All states once coverage strategy

This method iterates over all different paths in workflow graph G from its entry nodes *Initial(G)*. The different term in this strategy is referring to the set of nodes contained between paths. Let this set of paths in G be Paths(G). Any P1, P2 in Paths(G) differ by at least one node. Note that this method does not consider P1 and P2 as different paths if one cycles a different number of times through loops. Intuitively, this method tries to cover all nodes (activities) of the workflow at least once.

After gathering the set of all different paths using Depth-First-Search (DFS), the SMT solver (i.e., the external tool that solves the constraints accumulated on a path, in our case, Z3) is asked to find a test data for each path. The results are then displayed as in the figure below:



Pin_1:out_pi	Pin_1:local_r	Pin_1:local_p	Pin_1:expect	Pin_1:actual_	Pin_1:local_t	Pin_1:local_t	Main_1:loan	Main_1:term	Main_1:loan	Main_1:term	Main_1:obta	Main_1:pin_
							999	4				
							999	5				
							98712	4				
							16616	5				
				[0]=1234 ;			100000	4				
				[0]=1234 ;			100000	5				
				[0]=1234 ;			100000					
				[0]=1234 ;			100000	4				
				[0]=1234 ;			100000	5				
				[0]=1234 ;			100000					
				[0]=1235 ;			100000	4				
				[0]=1235 ;			100000	5				
				[0]=1235 ;			100000					
				[0]=1235 ;			100000	4				
				[0]=1235 ;			100000	5				
				[0]=1235 ;			100000					

```
// **Main loop** pseudocode
// First, getting all the paths that visit each node at most once in each of the output paths P
setOfPaths =  getAllPaths(G, currNode = G.entry_node, currPath = [] , visitedNodes = set(), setOfPaths)
for each P in setOfPaths
        // Get all the constraints needed to satisfy path P in graph G
        smt_constraints = getPathConstraints(P)
        // If all the branch constraints on the path are solvable, stream out the model obtained variables
        if SMTSolver(smt_contraints) has solution S:
                StreamOut(S)

// Getting all the constraints along a path pseudocode
getPathConstraints(P)
   // Hold a list of conditions that needs to be satisfied for the given path
  conditions_smt = []
   // Iterate over the path nodes
   for nodeIndex in |P.nodes| - 1
        node = P[nodeIndex]
        // If the node is branch, the constraint to jump from this node to the next in the path must be
        added
        if node.type == BRANCH:
                nextNode = P[nodeIndex + 1]
                // Check if the branch condition should be True or False to satisfy the condition
                evalExpected = True If node.nextNode(True)  == nextNode else False
                // Now add the condition specified by the node and transition
                If evalExpected == True:
                        conditions_smt.add(node.condition)
                else:
                        conditions_smt.add(Not(node.condition)
  // Solve the list of conditions and if solvable, stream out the solution variables
  S = SMTSolver.solve(conditions_smt)
   If S is valid:
        StreamOut(S)
```

// The pseudocode for getting all the paths in the workflow graph G, located in WorkflowGraph.getAllPaths

getAllPaths(G, currNode, currPath, visitedNodes, outAllPaths)

```
    // Add this node to the current path instance and mark as visited
    currPath.add(currNode)
    visitedNodes.add(currNode)

    // If the current node has no successors, leaf node, then just append the in progress-built path
    // to the output lists
    if currNode.next is None:
        outAllPaths.append(currPath)
    else:
        anyValidSuccessor = False
        // Branch the paths for each unvisited successor of this node in the workflow graph G
        for each successor S in currNode.next:
            if S not in visitedNodes:
                anyValidSuccessor = True
                getAllPaths(G, S, currPath + {S}, visitedNodes, outAllPaths)

        // It could happen that there is no unvisited node in this path, so just add it and exit
        If anyValidSuccessor == False:
            outAllPaths.append(currPath)
```

4. 2 Symbolic BFS/DFS

This method is different from the previous one in the sense that it explores exhaustively the graph G, including the loops. Thus, this method is more complex and gives better (complete) tests results, but with a higher computational cost.

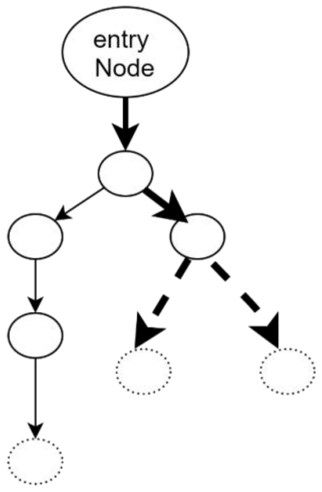
An execution path in this pure symbolic model is abstracted in the code in class *SMTPath*. When a branching decision occurs, the current *SMTPath* executed inside executor, *Curr_path*, is cloned and split in two: one taking the *True* and one taking the *False* branch. We add to the future worklist the feasible paths only, after computing their priority and set up the next node to execute according to the considered branch.

It is important to note that for one of the paths the *data_store* instance is also duplicated. This ensures that both operate on different data context values. This is also beneficial to parallelization since in the future, branches could be executed in parallel. The pseudocode of the operations is given below. Note that the execution style presented is BFS, i.e., after a branch both branches are added to the worklist according to the priority and exit. Then, the next top priority path continuation on the frontier is chosen (see figure below). However, there is also the possibility to run in a DFS style by continuing the current *True* branch if feasible and just add the other branch on the worklist. Or, even better, continue as direct continuation on the most promising path. The current implementation in the source code is by default on DFS since it was faster for our example than BFS.

Frontier nodes are discontinued circles, while the rest are visited.

The bolded path is the current Path. In a **BFS** style, both new frontier nodes are added to thepriority queue worklist, stop here and at the next iteration it will choose the top one (by priority) - possibily even the unexplored node in the left side of the workflow graph .

In **DFS** style, it will add one of the two new nodes in the worklist, but continue to execute on the other nodes as a continuation.



```
// Pseudocode of the main loop located in DFSSymbolicSolverStrategy.solve
// Init the start exploration path with the entry node of the workflow graph
start_path = SMTPath([G.entry_node], priority = MAX_PRIORITY)
// Add this to the current Worklist
W.add(start_path)
While W.notEmpty():
        currPath = W.extract()
        ExecutePath(currPath, W)


// Pseudocode of ExecutePath
ExecutePath(path : SMTPath,  W : Worklist)
  // Each path stores the next node to execute (initial is the entry node of the path)
  currNode = path.nextNodeToExecute


  // First, add all the assertions given by the DataStore objects restrictions given by annotations
  path.conditions_smt =  DataStoreTemplate.getVariablesAssertions()


  // The node will advance to the end of the Workflow graph G at each iteration of the path
  While currNode != None:
        // if current node is not a branch type, just execute it and advance to the next node in the
workflow
        if currNode.type != BRANCH:
                WorkflowExecutor.Execute(currNode)
                currNode = currNode.next()
        // If the current node is a branch,
        Else:
                // If branch node but there is no symbolic variable used in its condition expression
                //, then just advance
                If not HasSymbolicVariables(currNode.condition )
```

```
            // Get the result of the evaluation and advance
            Result = WorkflowExecutor.Execute(currNode)
            currNode.advance(Result)
            continue

        // Node has symbolic variables, build the continuation of the current path on both
        // True and False branches with the next nodes and SMT assertions needed
        newPath_onTrue = SMTPath(nodes=currPath.nodes + currNode.nextNode(True),
            conditions_smt= currPath. conditions_smt + currNode.condition)
        newPath_onFalse = SMTPath(nodes=currPath.nodes + currNode.nextNode(False),
            conditions_smt= currPath. conditions_smt + Not(currNode.condition))

        // Then check which of them are feasible, prioritize, set the next starting nodes, clone the
        data store instances such that they could work in parallel, and add them to the worklist
        solvableNewPaths = []
        if SMTSolver(newPath_onTrue) has solution:
            solvableNewPaths.add(newPath_onTrue)
        If SMTSolver(newPath_onFalse) has solution:
            solvableNewPaths.add(newPath_onFalse)
        for newPath in solvableNewPaths:
            newPath.scorePath()
            newPath.data_store = currPath.data_store.clone()
            newPath.nextNodeToExecute = currNode.nextNode(True if newPath ==
            newPath_onTrue else False)
            W.add(newPath_onFalse)
        Break
    // If get at the end of the path stream out the model variables
    StreamOut(currPath)
```

4.3 Concolic method

While the pure symbolic method defined above can exhaustively explore paths and produce in theory particularly good coverage results, there are some workflows that would take too much computational effort. Thus, we use the state-of-the-art theory and leverage concolic execution for those cases. Such an optimized method is the SAGE, Whitebox fuzzing by Patrice Godefroid[1]. The pseudocode of the original method is given below.

```
Search(inputSeed){
  inputSeed.bound = 0;
  workList = {inputSeed};
  Run&Check(inputSeed);
  while (workList not empty) {//new children
    input = PickFirstItem(workList);
    childInputs = ExpandExecution(input);
    while (childInputs not empty) {
      newInput = PickOneItem(childInputs);
      Run&Check(newInput);
      Score(newInput);
      workList = workList + newInput;
    }
  }
}
```

```
ExpandExecution(input) {
  childInputs = {};
  // symbolically execute (program,input)
  PC = ComputePathConstraint(input);
  for (j=input.bound; j < |PC|; j++) {
    if((PC[0..(j-1)] and not(PC[j]))
                      has a solution I){
      newInput = input + I;
      newInput.bound = j;
      childInputs = childInputs + newInput;
  }
  return childInputs;
}
```

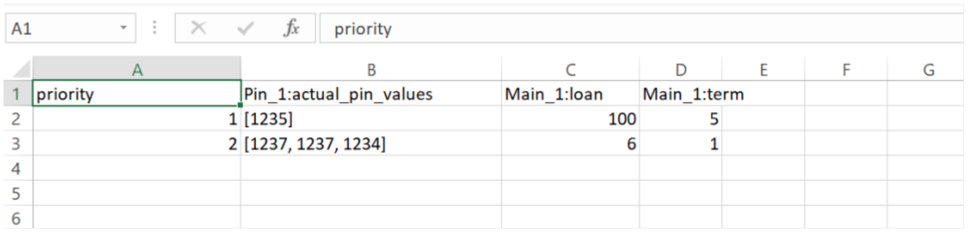
We adapt the methods to be used in our context of RPA workflows, in class `ConcolicSolverStrategy` and the main observations / modifications are the following.

We do not need to Run&Check then score the input as in the original method. The reason is that the original does not know which code was touched by the new code, but we do since we have **full access to**

[1] https://patricegodefroid.github.io/public_psfiles/cacm2012.pdf

the graph! So, this is an important for us as observed in `ConcolicSolverStrategy.generateNewInputs`, which is the correspondent for the *ExpandInputs* in the original psecudocode shown above.

Part 1 reading and generating seeds: The `ConcolicSolverStrategy.solve` function first reads the **input seeds file** (given as input parameter –**seedsFile),** a csv containing all the initial set of seeds given by a human or automatic method to the Fuzzer as a hint to start better in searching things. An example from BankLoadWithPin:



A1 | priority

	A	B	C	D	E	F	G
1	priority	Pin_1:actual_pin_values	Main_1:loan	Main_1:term			
2	1	[1235]	100	5			
3	2	[1237, 1237, 1234]	6	1			
4							
5							
6							

The priority column is given as a hint, i.e., what is the priority (lower value means higher priority) to start with that particular item in testing.

Then, user can also request a number of random test values to start with, given by parameter -`numRandomGeneratedSeeds`**.** Both are optional, but the sum of both must be > 0 such that the model has at least one item to start with. All these inputs are added in the worklist W, which is executed in the function mentioned.

Part 2 main loop of concolic, implemented in the same ConcolicSolverStrategy.solve function

The loop:

A. Takes the next top priority inputSeed in W,

B. Injects the values from the inputSeed to the DataStore current context

C**.** And executes the model in a concolic strategy: **it does not change the branch conditions at runtime. Just goes through the model from begin to end and executes the model.**

At this time, as observed in function addNewBranchLevel, on the concolic case, it stores at each branching decision two things:

(a) what was the branch decision, as a Boolean, taken or not?

(b) what is the inverse SMT condition to get to the other branch?

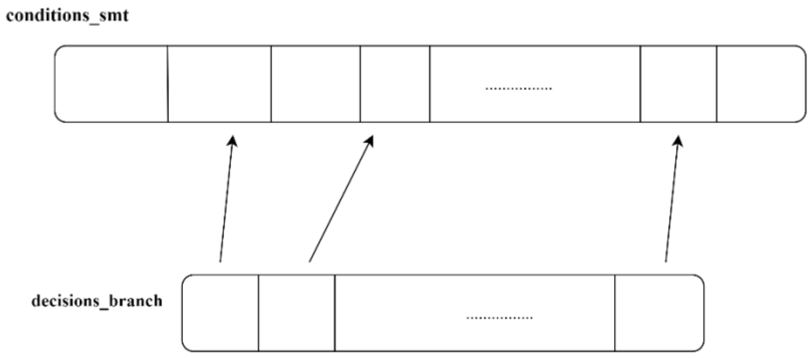
Given an executed path *PathExecuted* with all its assumptions for the model in *PathExecuted.conditions_smt,* the concolic case, the output of this step will be additionally a dictionary containing the index of the SMT in *PathExecuted.conditions_smt* and two value pairs: was the branch taken or not, i.e., was the condition statement True?, then SMT inverse condition to get the the other branch.

Why is indexing needed? Because not all the conditions inside *PathExecuted.conditions_smt* are used as branch decisions in symbolic expressions! For instance, these could contain instead conditions related to the range of variables, as given for annotations!

```
// Pseudocode for executing an SMTPath in concolic mode:
ExecutePath(path : SMTPath)
  currNode = path.entryNode
  indexBranchAlongPath = 0
  // First, add all the assertions given by the DataStore objects restrictions given by annotations
  path.conditions_smt = DataStoreTemplate.getVariablesAssertions()
  While currNode is not None:
        // if current node is not a branch type, just execute it and advance to the next node in the workflow
        if currNode.type != BRANCH:
              WorkflowExecutor.Execute(currNode)
              currNode = currNode.next()
        // If the current node is a branch,
        Else:
              // Get the result of the evaluation
              Result = WorkflowExecutor.Execute(currNode)

              // If branch node but there is no symbolic variable used in its condition expression
              //, then just advance
              If not HasSymbolicVariables(currNode.condition )
                    currNode.advance(Result)
                    continue

              // Store what is assertion to take again the same branch in a future run and
              // what is the condition to go to the other branch path at this point
              If Result is True:
                    Taken_assert = currNode.condition
                    Inverse_assert = Not(currNode.condition)
                Else:
                    Taken_assert = Not(currNode.condition)
                    Inverse_assert = currNode.condition

                // Add the current path assertion for this branch to the set of decisions
                path.conditions_smt.add(Taken_assert)
                // And the inversed branch as option
                path.decisions_branch.Add(indexBranchAlongPath, Result, InverseAssert)
                indexBranchAlongPath += 1
                // And advance the node iteration in the resulted branch
                currNode.advance(Result)

// If get at the end of the path stream out the model variables
Stream Out(currPath)

// Main loop pseudocode
While not W.empty():
        // Extract the top priority Seed
        seedToUse = W.extract()
        // Fill in an SMTPath data structure with this seed
        newPath = SMTPath(seedToUse)
        // Execute the path in concolic mode
        ExecutePath(newPath)
        // Now the path contains the branching decision points so new inputs can be generated (see Next
        pseucode and sub-section) and added to the worklist W
        GenerateNewInputs(newPath, W)
```

Part 2. Generating new inputs seeds

After a *PathExecuted* is generated and finished, it contains the conditions_smt and decisions_branch dictionaries, as specified before. They look like in the figure below:



There is an **injection from decisions_branch to conditions_smt**, as specified in Part 1. All indices in decisions_branch are part of conditions_smt set, but not the inverse, since conditions_smt could store assertions that are not related to a branch decisions. As mentioned above, each item *Decision* \in *decision_Branch* has two members {*taken* : bool, *inversedSMTCondition :z3Ast*}.

Having this setup, the role of function ConcolicSolverStrategy.generateNewInputs is to generate new inputs seeds out of *PathExecuted.* The pseucodocode execute is the following:

GenerateNewInputs(PathExecuted)

```
    // Init the output, it will be a list of generated new inputs of structure InputsSeeds, a tuple of –
    content (dictionary mapping from variabile name to value), bound (index of the bound in the tree,
    check the above original pseucodocode).

    InputsGenerated : List[InputsSeeds] = []

    // Init a local solver

    S = SMTSolver()

    Corr = PathExecuted.decision_branch.CorrespondenceIndices


    // Init two iterators, one over indices over each Smt condition

    iter_origCond = 0 in PathExecuted.conditions_smt

    // the other one in the correspondences from decisionsBranch to conditions_smt

    iter_decisionBranch = 0 in Corr
```

```
while iter_origCond < len(PathExecuted.conditions_smt):
 // If not yet at a decision branch, just add the original assertion to the solver
        If iter_origCond < Corr[iter_decisionBranch]
                S.add_assertion(PathExecuted.conditions_smt[iter_origCond])
                iter_origCond += 1

        // If we are at a decision branch
        Else if iter_origCond == Corr[iter_decisionBranch]:
                // Save the SMT solver state on stack
                S.push()
                // Add the inversed condition to try the other branch
                S.add(PathExecuted.decision_branch[iter_decisionBranch].inversedSMTConditio
                n)
                // If the model has a solution output a new input seed, note that its bound is one
                step forward to avoid backtracking, like in the original SAGE algorithm.
                If IsSatisfiable(S):
                        inputContent = S.solution()
                        InputsGenerated = InputsGenerated U InputSeed{content=
                        inputContent, bound = PathExecuted.bound + 1, score =
                        scoreInput(content)}

                // Now pop the solver state back, and add the original condition to continue
                correctly along the decisions branching paths
                S.pop()
                S.add_assertion(PathExecuted.conditions_smt[iter_origCond])
                iter_origCond += 1
                iter_decisionBranch += 1
```

4.4 Worklist implementation and Extension with priorities

The priority queue worklist W used in the methods described above, is based on:

A) In symbolic execution: comparing the priorities of the next nodes available in W
B) In concolic execution: based on the priority of the input seeds added to W.

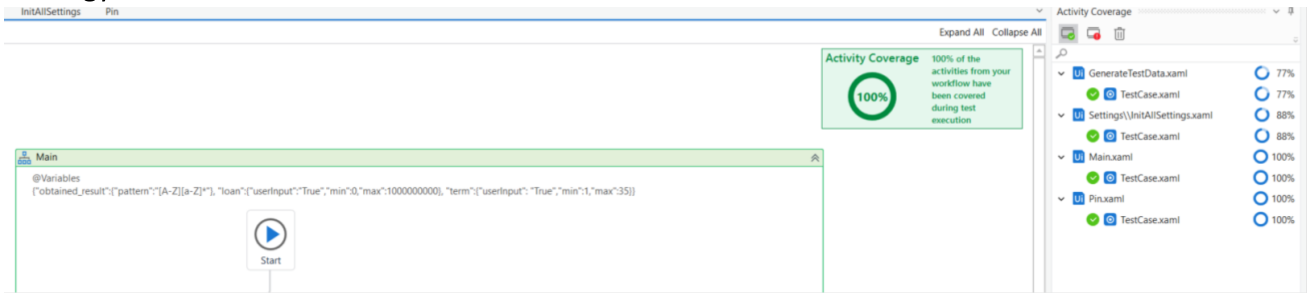
In both cases, the default strategy used is the level in the tree where the branching point occurs when generating the next node or input seed is generated. Such, we promote paths at the beginning of the graph instead of exploring in deep the loops.

But this is a default strategy only. An interested developer could further implement other methods by inheriting the two classes in the diagram and rewrite its own *scoreNewConcolicInput* respectively *scoreNewSymbolicPath*.

5. Evaluation

Model tested	**Solver strategy**	**Path coverage (%)**	**Fuzzer execution time (seconds)**
SimpleBankLoanWithPin	STRATEGY_OFFLINE_ALL	100%	3.505171
SimpleBankLoanWithPin	STRATEGY_DFS	100%	2.557874
SimpleBankLoanWithPin	STRATEGY_CONCOLIC	100%	3.187291

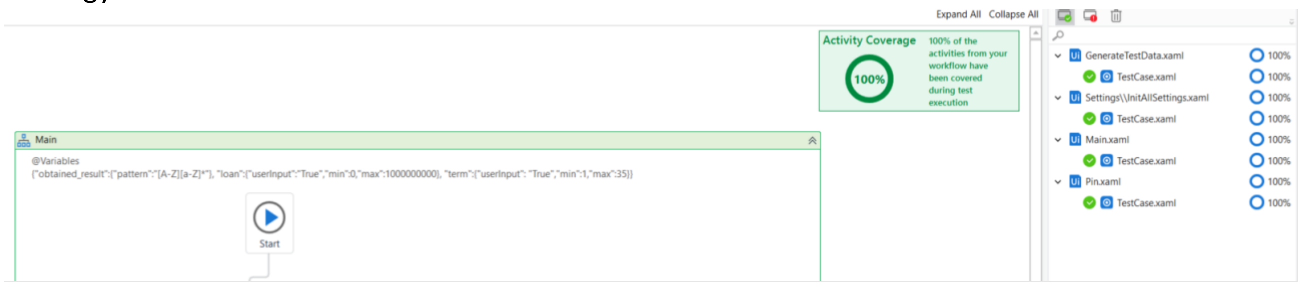
Strategy Offline all



```
@Variables
{"obtained_result":{"pattern":"[A-Z][a-Z]*"}, "loan":{"userInput":"True","min":0,"max":1000000000}, "term":{"userInput": "True","min":1,"max":35}}
```

GenerateTestData.xaml	77%
TestCase.xaml	77%
Settings\\InitAllSettings.xaml	88%
TestCase.xaml	88%
Main.xaml	100%
TestCase.xaml	100%
Pin.xaml	100%
TestCase.xaml	100%

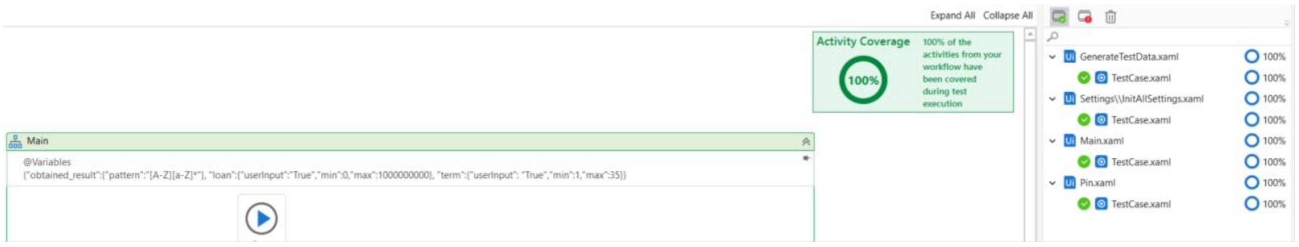
Strategy DFS


Expand All
Collapse All
Activity Coverage
100%
100% of the activities from your workflow have been covered during test execution
Main
Start

```
@Variables
{"obtained_result":{"pattern":"[A-Z][a-Z]*"}, "loan":{"userInput":"True","min":0,"max":1000000000}, "term":{"userInput": "True","min":1,"max":35}}
```

GenerateTestData.xaml	100%
TestCase.xaml	100%
Settings\\InitAllSettings.xaml	100%
TestCase.xaml	100%
Main.xaml	100%
TestCase.xaml	100%
Pin.xaml	100%
TestCase.xaml	100%

Strategy Concolic



```
@Variables
{"obtained_result":{"pattern":"[A-Z][a-Z]*"}, "loan":{"userInput":"True","min":0,"max":1000000000}, "term":{"userInput": "True","min":1,"max":35}}
```

6. A couple of ideas for extending the work with new methods

At the end we provide a couple of ideas for extending the current prototype:

- Parallelization of SMTPath execution in all strategies; this is possible as explained before since each has its own DataStore context
- Implementation of more strategies and priorities (see the literature review in supplementary material)
- For the concolic case, since we have access to the full graph, we can understand which of the paths in the worklist are similar and prune them. This is a huge optimization possibility.
- DataTables types cannot be currently generated symbolically or in concolic mode. They can come at user inputs but cannot be symbolically evaluated. It will be nice in future work to generate FULL data tables, and it is doable!
- Plans for real-time robot interaction/feedback for methods in Section 3 by running the concolic method and getting back branches from the robot at every step instead of relying on the fixed graph only.