Artifact - Historia: Refuting Callback Reachability with Message-History Logics

ACM Reference Format:
. 2023. Artifact - Historia: Refuting Callback Reachability with Message-History Logics. 1, 1 (July 2023), 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This document explains the artifact for the Historia paper [1]. The goal of this document is to first give a set of instructions for reproducing the experimental results and then give a technical explanation of how the implementation connects to the technical contributions. For inputs, this artifact takes a compiled Android application in the form of an APK file, a location in the application for the assertion, and a CBCFTL specification of realizable message histories. Outputs are either “safe” or “alarm” in which case an abstract message history witnessing the alarm will be available.

2 PREREQUISITES - RUNNING THE HISTORIA DOCKER CONTAINER

We have configured the experiments to be run within a Docker container provided with this artifact. This Docker file may be found in the root directory of this archive and is labeled `historia.docker`. Please follow the instructions to install docker from https://docs.docker.com/engine/install/.

Importing the docker container can be done with the following command.

```
docker import historia.docker
```

The docker container may be run with the following command. [**TODO:** *expose web port and swap with jupyter command*]

```
docker run --memory="8G" --memory-swap="8G" --rm -it historia bash
```

All subsequent steps may be done through the web interface at a URL printed by the terminal window the docker command was run. This URL should start with http://localhost:8080 and contain an encoded security token. Opening this URL should show a Jupyter notebook. [**TODO:** *screenshot of jupyter notebook*]

System Requirements. We have split the instructions so that a subset of the experiments may be run on a reasonable laptop and included full instructions if access to a server is available. The instructions labeled "minimal resources" may be run on a reasonably modern laptop that has 8GB of ram in addition to ram used by other processes.

Instructions labeled with "full resources" can be run on more powerful systems. We used a server with a AMD EPYC 7763 64-Core Processor and 256GB of ram. Running the full instructions on lower end systems will result in timeouts

Author’s address:

and out of memory errors. Please note that sometimes the process will be killed with no memory related error message if not enough ram is available, this is simply a JVM/JNI limitation.

3 REPRODUCING THE HISTORIA RESULTS

Each subsection here corresponds to a table in the evaluation section. We have labeled the subsections first with the research question (**RQ1** or **RQ2**), the table number (Table 1, Table 2, Table 3), and finally, the resource requirements as described earlier (minimal resources or full resources).

3.1 RQ1 - Table 1 - minimal resources

The experiments in Table 1 are run as a Scala unit tests defined in the file:
`src/test/scala/edu/colorado/plv/bounder/symbolicexecutor/Experiments.scala`.

Each benchmark is labeled by `test([description])` where [description] has a row number and short english description. The source code for each benchmark is a string stored in a variable named `src`. This source code is compiled into the APK automatically when running the unit test.

The set of CBCFTL specifications for each row are defined by row in the `ExperimentSpecs` object within `Experiments.scala`. These specifications may be found in `src/main/scala/edu/colorado/plv/bounder/lifestate/Specification.scala`.

To run the experiments for the first table, open the terminal in the Jupyter notebook listed earlier and run the following commands:

```
cd home/bounder
bash runExperiments.sh
```

The output will be printed to the screen in a log format. Each row of Table 1 may be found labeled by "Row" followed by the row number and version (i.e. "bug" or "fix"). For example, the buggy version of `getAct`[??] is "Row 1 bug" and the fixed version is "Row 1 fix".

Below is an explanation of each column in the table:

(1) Pattern cb,ret - The number of callbacks and returns in the bug pattern: Integer labeled "cbSize".
(2) Pattern ci - the number of callins in the bug pattern: Integer labeled "syntCi".
(3) Historia specs - number of specs written for the benchmark. Integer labeled "spec count". The specific specs may be found in `Experiments.scala` and `Specification.scala` as described earlier.
(4) Historia res - The result of the analysis: There will be several rows of text displaying the data from the table. The result of verification labeled with "actual:" and may say "Witnessed" (ⓘ), "Timeout" (🕓), or "Proven" (⊘).

For reference, it also prints the expected result after "expected:". For the rest of the columns in Table 1 in order:
[**TODO:** *explain message counts etc*]
[**TODO:** *SM: explain how to compare flowdroid and infer*]

3.2 RQ2 - Table 2 - minimal resources

3.3 RQ2 - Table 3 - minimal resources

3.4 RQ1 - Table 1 - full resources

[**TODO:** *–Full resources*]

3.5 Running RQ1 (full)

[**TODO:**]

3.6 Running RQ2 (full)

[**TODO:**]

[**TODO:** *third priority is comparison with infer/flow*]

4 RUNNING AND INTERPRETING HISTORIA ON CUSTOM INPUTS

[**TODO:** *second priority*]

Historia may be run on arbitrary Android applications as long as an APK can be compiled in debug mode. This is usually accomplished using the command `./gradlew assembleDebug` but will vary from application to application. A location and safety property must be chosen ahead of time. However, we recommend only writing CBCFTL specifications as needed.

Running Historia Through Jupyter. The recommended way to run Historia is using a Jupyter notebook. This allows the inputs to the tool to be defined using the Scala data structures for input rather than JSON.

[**TODO:** *give sample app to walk through this process*]

[**TODO:** *explain this process completely*]

[**TODO:** *explain counter examples*]

5 DEVELOPING FOR HISTORIA

[**TODO:** *fourth priority*]

In this section, we explain the implementation of each technical contribution in Historia.

5.1 Running Unit Tests

[**TODO:** *this may need to go in later connecting formalism section*]

5.2 Application-Only Control-Flow Graph

[**TODO:**]

5.3 Message-History Program Logic (MHPL)

[**TODO:**]

5.4 Callback Control-Flow Temporal Logic (CBCFTL)

[**TODO:**]

5.5 Combining Abstract Message Histories with Callback Control Flow

[**TODO:**]

5.6 Optimizations

[**TODO:**]

REFERENCES

[1] Shawn Meier, Sergio Mover, Gowtham Kaki, and Bor-Yuh Evan Chang. 2023. Historia: Refuting Callback Reachability with Message-History Logics.. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.