# Analysis Tools for the VyPR Performance Analysis Framework for Python


Joshua Heneage Dawes University of Manchester, Manchester, UK
CERN, Geneva, Switzerland

joshua.dawes@cern.ch

Marta Han University of Zagreb, Zagreb, Croatia
CERN, Geneva, Switzerland

Giles Reger University of Manchester, Manchester, UK

Giovanni Franzoni CERN, Geneva, Switzerland

Andreas Pfeiffer CERN, Geneva, Switzerland



Computing in High Energy Physics, 2019

# Analysis by Specification

Using state and time constraints.

Efficient monitoring to check whether the program behaves as described.

Description of expected behaviour of individual functions written by developer → Instrumentation → Asynchronous monitoring

To check behaviour, we take measurements at runtime, but we take as few as possible.

The University of Manchester

# Description by Example

Select points of interest at runtime

```
Forall(
  s = changes('a')
).Check(
  lambda s : (
    timeBetween(s, s.next_call('f').result())._in([0, 1])
  )
)
```

*Every time a changes, the time between that change and the end of the next call to f should be no more than 1 second.*

Defines the rule to check at each of these points of interest

CERN MANCHESTER 1824 The University of Manchester

# Instrumentation

For web services, VyPR's current main use case, instrumentation is performed between deployment and service start.

Original file is kept but renamed to force imports to use the instrumented bytecode.

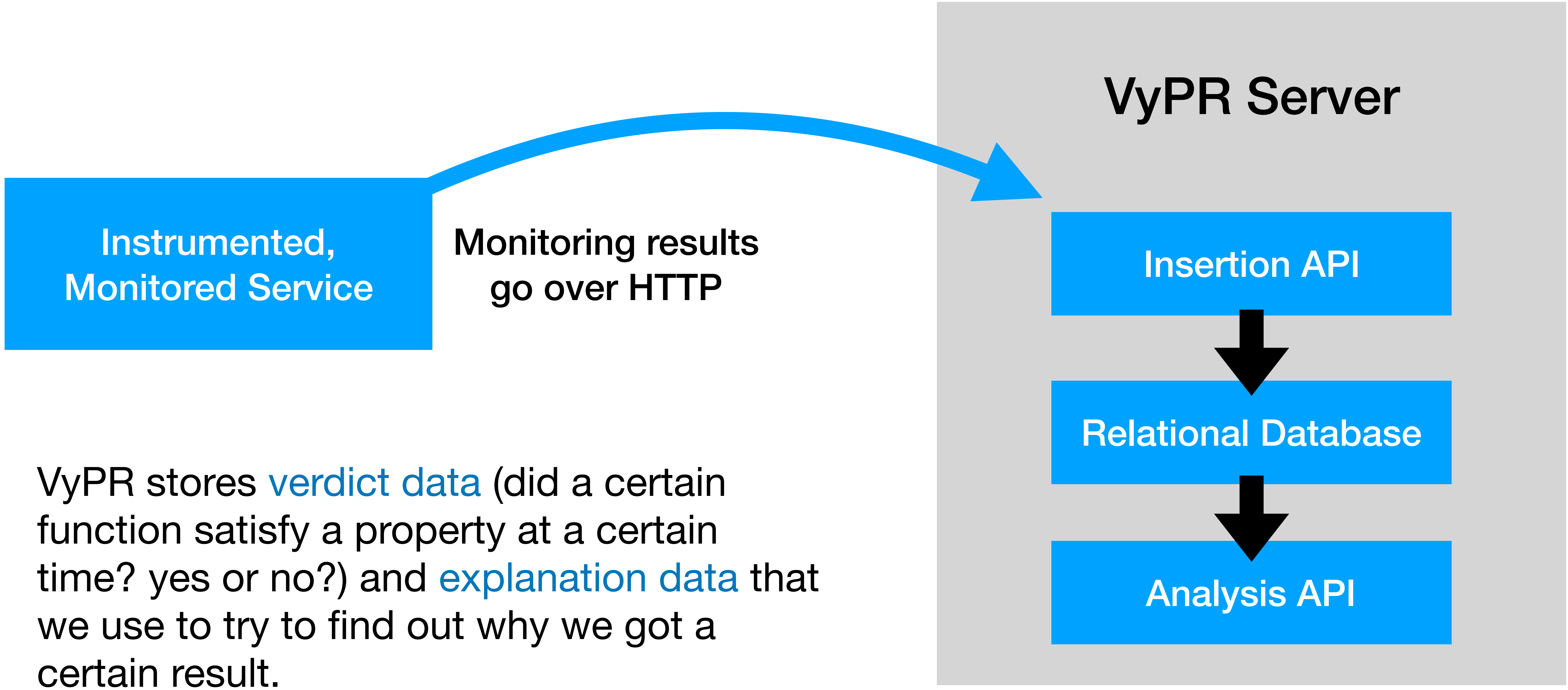
Source Code of Monitored Function → AST modification - automatic instrumentation → Compilation to Python bytecode → Enough information to check behaviour

VyPR derives an *augmented* control flow graph and uses this to perform static analysis, which allows conservative instrumentation.

Instrumentation is performed by adding ASTs of instrumentation code to the AST representation of the program, and then compiling to bytecode.

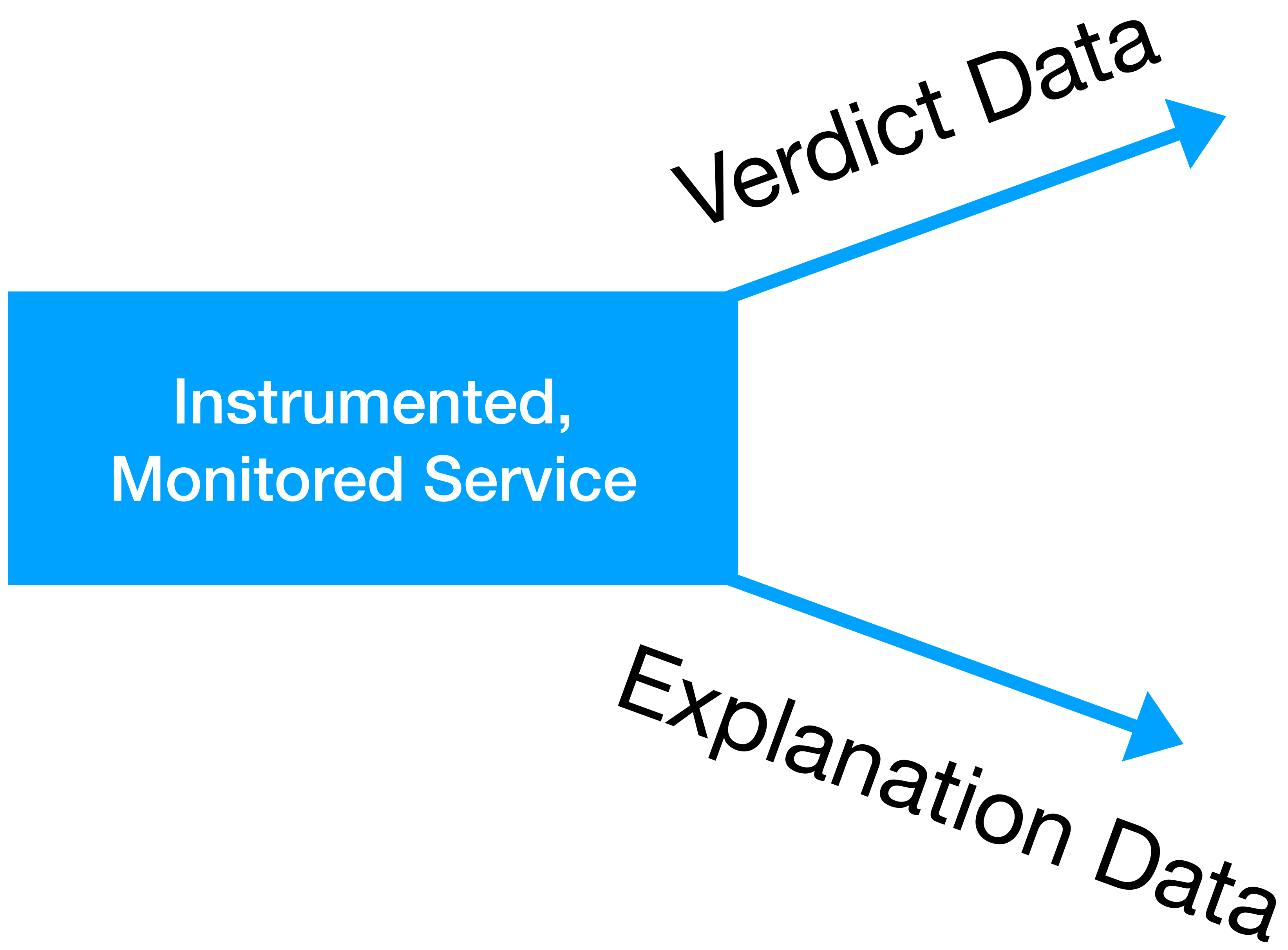
CERN MANCHESTER 1824 The University of Manchester

# Collecting Monitoring Data




VyPR stores verdict data (did a certain function satisfy a property at a certain time? yes or no?) and explanation data that we use to try to find out why we got a certain result.

# How does the data look?




### Satisfaction/Violation

We record whether things went well, and when.

### Which part of our description was violated?

We record the constraint that was the first to tell us something was wrong.

### Variable values at key points

If we place constraints over function calls, we might care about the values present before the call.

### Program paths

We record the sequence of branches taken and map observations to the previous satisfied branching condition.

### Function call stack

We store enough information to be able to reconstruct the call stack of all functions whose behaviour was described.

# VyPR Analysis

Object-oriented library for Python.

Methods defined to make common tasks (that require complex queries) straightforward.

Powerful internals currently help the discovery of root causes using very little code.

Analysis library communicates with a central verdict server.

# Determining Problematic Control-Flow with VyPR's Analysis Library

```
Forall(c = calls('func')).\
Check(lambda c : (
  c.duration()._in([0, 0.01])
))
```

```
import VyPRAnalysis as analysis
import VyPRAnalysis.orm.operations as ops
```

```
analysis.set_config_file("VyPRAnalysis/config.json")
```

Connect to a verdict server

```
functions = analysis.list_functions()
f = functions[0]
```

Fix a function and a property over that function

```
verdicts = f.get_verdicts()
observations = [
  verdicts[0].get_observations()[0],
  verdicts[1].get_observations()[0]
]
obs_collection = ops.ObservationCollection(
  observations
)
```

Get a list of observations that were required to check the property

```
path_collection = obs_collection.to_paths()
path_collection.show_critical_points_in_file(
  filename="critical_points"
)
```

Determine the points in control-flow at which paths leading to those observations diverged.

# Sample Output

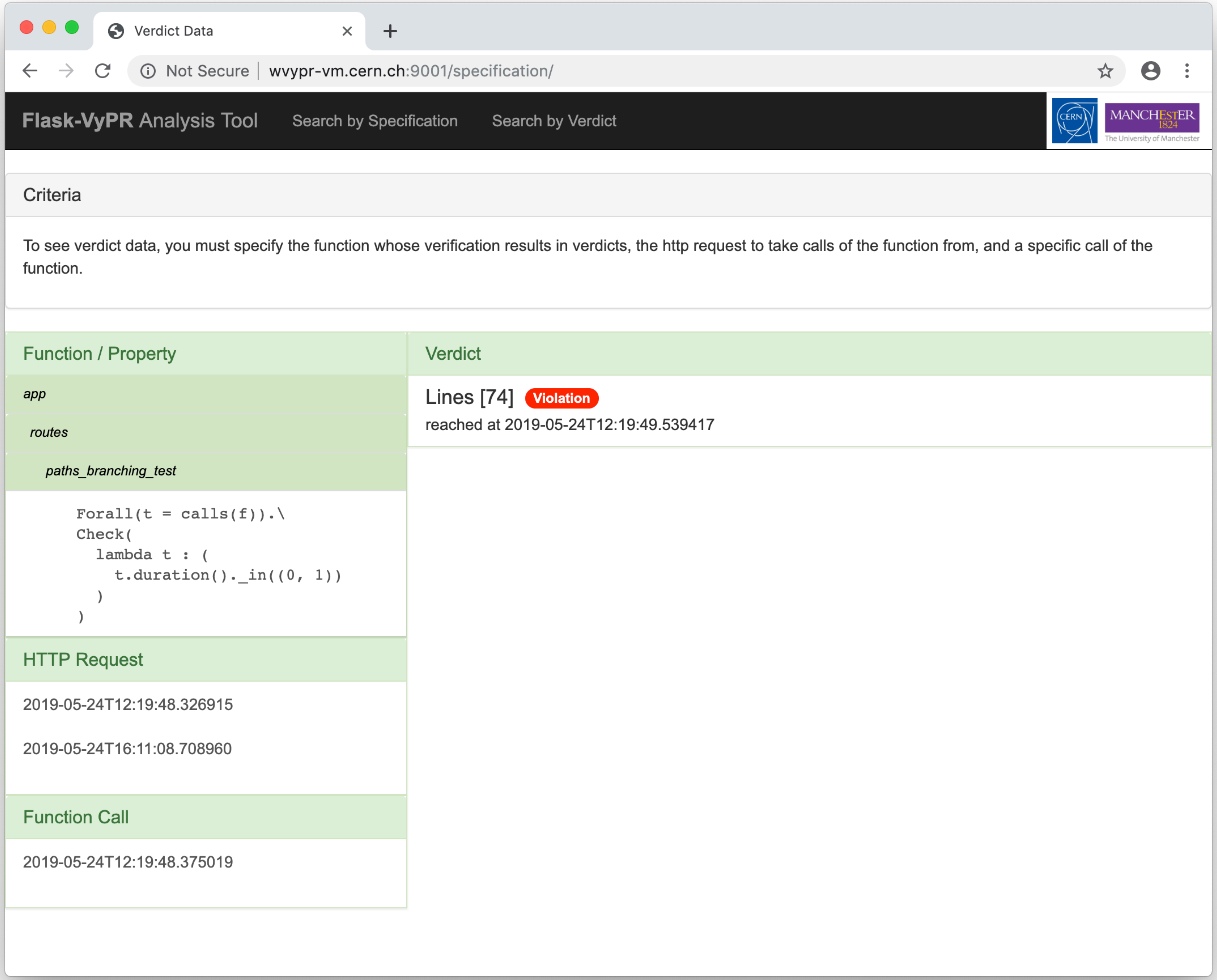
```
Forall(c = calls('func')).\
Check(lambda c : (
  c.duration()._in([0, 0.01])
))
```

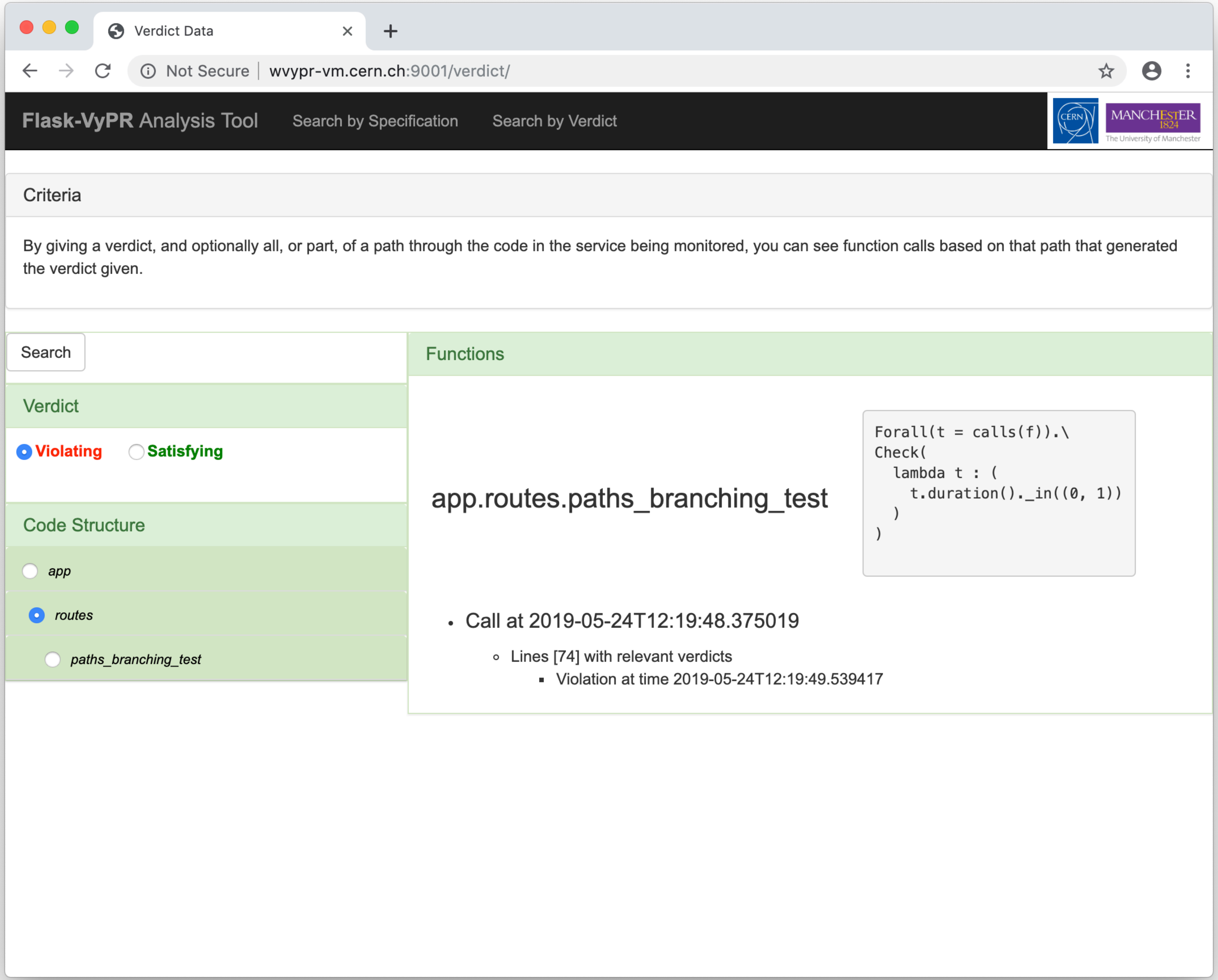
```
Critical points in code for satisfying paths:
46       g.usage.log("\tConnected to Destination Database.")
47
48  *    if self.tag_in_destination:
49         g.usage.log("\tDestination Tag '%s' found." % […])


Critical points in code for violating paths:
46       g.usage.log("\tConnected to Destination Database.")
47
48  *    if self.tag_in_destination:
49         g.usage.log("\tDestination Tag '%s' found." % […])
```

The University of Manchester

# A Web Application for Visual Analysis

***Prototype stage***


Verdict Data
Not Secure | wvypr-vm.cern.ch:9001/specification/
Flask-VyPR Analysis Tool
Search by Specification
Search by Verdict
CERN
MANCHESTER 1824
The University of Manchester
Criteria
To see verdict data, you must specify the function whose verification results in verdicts, the http request to take calls of the function from, and a specific call of the function.
Function / Property
app
routes
paths_branching_test
Forall(t = calls(f)).\
Check(
lambda t : (
t.duration()._in((0, 1))
)
)
HTTP Request
2019-05-24T12:19:48.326915
2019-05-24T16:11:08.708960
Function Call
2019-05-24T12:19:48.375019
Verdict
Lines [74]
Violation
reached at 2019-05-24T12:19:49.539417


Verdict Data
Not Secure | wvypr-vm.cern.ch:9001/verdict/
Flask-VyPR Analysis Tool
Search by Specification
Search by Verdict
MANCHESTER 1824
The University of Manchester
Criteria
By giving a verdict, and optionally all, or part, of a path through the code in the service being monitored, you can see function calls based on that path that generated the verdict given.
Search
Verdict
Violating
Satisfying
Code Structure
app
routes
paths_branching_test
Functions
app.routes.paths_branching_test
Forall(t = calls(f)).\
Check(
lambda t : (
t.duration()._in((0, 1))
)
)
Call at 2019-05-24T12:19:48.375019
Lines [74] with relevant verdicts
Violation at time 2019-05-24T12:19:49.539417

# Application at CMS

2018 experiments with CMS' release service for alignment and calibrations showed unexpected performance drops.

*J H Dawes, G Reger, G Franzoni, A Pfeiffer, G Govi. VyPR2: A Framework for Runtime Verification of Web Services. TACAS 19.*

2019 experiments, with path analysis and state comparison, have shown:

1. The branch taken in one case (which depends on the data being uploaded) does not affect the performance. This is a good performance characteristic to know about.

2. The time required to perform a check for existence of some input mostly depends on the size of the input, with some fluctuation expected. This answers our question regarding for how much network latency was responsible.

VyPR performs well, even with the heavier explanation mode enabled.

# Goals

The work developers have to do to determine root causes of behaviour that disagrees with what's expected should be minimised.

Research for VyPR is aiming at removing as much developer involvement as possible from the root cause determination process.

VYPR

Publicly available - cern.ch/vypr

We are looking for new contributors, collaborators and applications:

joshua.dawes@cern.ch