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

Com computing in High Energy Physics, 2019
Analysis by Specification

Description of expected behaviour of individual functions written by developer

Using state and time constraints.

Instrumentation

Efficient monitoring to check whether the program behaves as described.

Asynchronous monitoring

To check behaviour, we take measurements at runtime, but we take as few as possible.
For all \( s = \text{changes}('a') \).

Check

\[
\lambda s : (\text{timeBetween}(s, s.\text{next\_call}('f').\text{result}()) \in [0, 1])
\]

Select points of interest at runtime

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
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.
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

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

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

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

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

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

- Connect to a verdict server
- Fix a function and a property over that function
- Get a list of observations that were required to check the property
- Determine the points in control-flow at which paths leading to those observations diverged.

Forall(c = calls('func')).\Check(lambda c : (c.duration() in([0, 0.01])))
Sample Output

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

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

For all \(c = \text{calls}(\text{\texttt{\textquoteleft}func\textquoteright})\).
Check(\lambda c : (c.duration() \in [0, 0.01]))
A Web Application for Visual Analysis

Prototype stage
<table>
<thead>
<tr>
<th>Function / Property</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>Lines <a href="#">74</a> Verification reached at 2019-05-24T12:19:49.539417</td>
</tr>
<tr>
<td>routes</td>
<td></td>
</tr>
<tr>
<td>paths_branching_test</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
Forall(t : calls(f)).
    Check(
        lambda t : {
            t.duration() in [0, 1]
        }
    )
```

**HTTP Request**

- 2019-05-24T16:11:08.708960

**Function Call**

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.

Code Structure
- app
- routes
- paths_branching_test

Functions

```
app.routes.paths_branching_test

  - 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.


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.
Publicly available - cern.ch/vypr

We are looking for new contributors, collaborators and applications:

joshua.dawes@cern.ch