Automated Performance Analysis for Python with VyPR

cern.ch/vypr

Joshua Heneage Dawes University of Manchester, Manchester, UK
CERN, Geneva, Switzerland
joshua.dawes@cern.ch

Giles Reger University of Manchester, Manchester, UK

Giovanni Franzoni CERN, Geneva, Switzerland

Andreas Pfeiffer CERN, Geneva, Switzerland

Fermilab - CMS Offline Software and Computing Week

Typical Analysis of Behaviour for Python

Checking the behaviour of programs requires a lot of manual effort.

We would start with deciding which constraints we want to check.

Then we would need to address instrumentation… manual insertion of profiling code.

Then manual offline inspection of the data obtained.

If constraints get more complex, manual instrumentation and offline inspection can easily become infeasible.

The General Idea of VyPR

VyPR reduces manual effort by allowing complex constraint description over single runs of functions, automatic instrumentation and efficient monitoring:

1. Describe behaviour of functions in your program via constraints over values and timing.

2. Use VyPR to automatically instrument and then monitor your program at runtime for agreement.

3. Use VyPR's analysis tools to help determine root causes of incorrect behaviour.

Our current development effort focuses on web services.

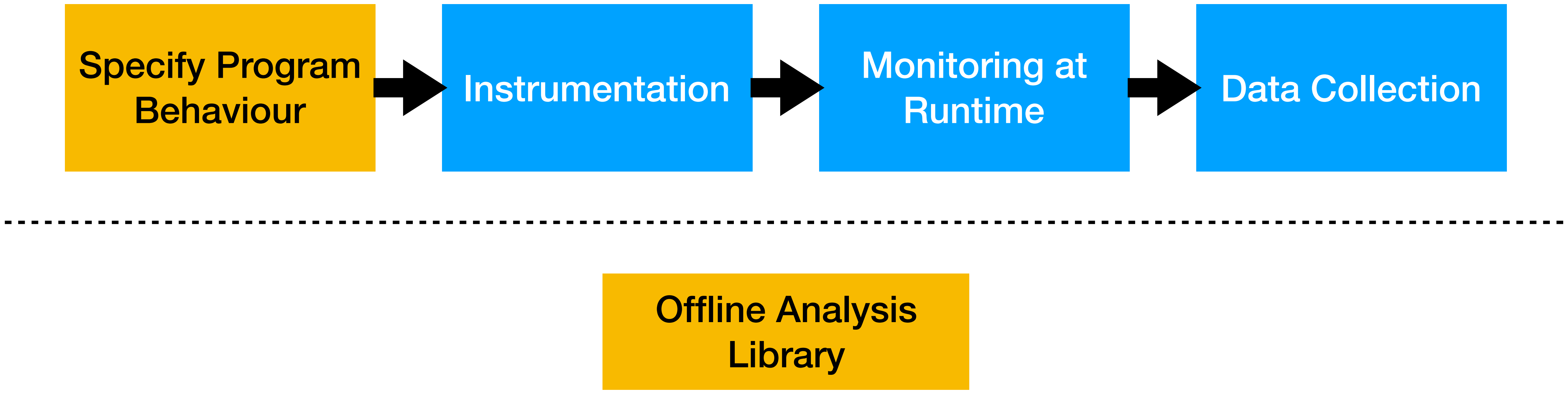
Overview of VyPR

Intuitive, “close to natural language” Python library

Automatic, aiming at minimising overhead at runtime

Automatic, sends data per function to a central server

Space efficient indexing on a remote server, ready for offline analysis



Minimal Intrusion

No change is required to the code a developer wants to monitor.

In applications so far, only a couple of lines are required to activate monitoring by VyPR.

All description is done in a separate configuration file.

VyPR has been shown to generalise well, being applied to Flask-based web services and a web service with entirely custom-written internals.

Research for VyPR

VyPR is one of the main research outputs of my PhD at CMS.

Everything described in this talk is supported by theoretical foundations, published in the papers:

J H Dawes, G Reger. Specification of State and Time Constraints for Runtime Verification of Functions. arXiv 2018.

J H Dawes, G Reger. Specification of temporal properties of functions for runtime verification. SAC 2019.

J H Dawes, G Reger, G Franzoni, A Pfeiffer, G Govi. VyPR2: A Framework for Runtime Verification of Python Web Services. TACAS 2019.

J H Dawes, G Reger. Explaining Violations of Properties in Control-Flow Temporal Logic. RV 2019.

J H Dawes, M Han, G Reger, G Franzoni, A Pfeiffer. Analysis Tools for the VyPR Performance Analysis Framework for Python. CHEP 2019.

Behaviour Description by Example

"Whenever **n** changes, require that no more than 1 second passes before the end of the next call to **f** is reached."

```
n = param
if n > 10:
  l = []
  for i in range(n):
    l.append(g(i**2))
else:
  l = []
f(l)
```

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

Explaining Failure

If your program fails to satisfy your specification, VyPR provides tools to help you determine why.

Problematic Paths - can a branch be isolated that causes problems?

Can we conclude that the branch taken doesn't matter?

Is there a function that often causes a problem with a computation when it is called on a path leading to it?

Problematic Values - is there a set of values that are causing a problem?

Inter-procedural Analysis - under development - are there problematic paths/values up/down the call chain?

Problematic Paths

```
  n = param
* if n > 10:
    l = []
    for i in range(n):
      l.append(g(i**2))
  else:
    l = []
  f(l)
```

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

If some program runs fail a specification and some satisfy it, VyPR provides the machinery to detect critical branching points.

Notable Application

Main experience - CMS Conditions Upload service used by calibrations experts working in PPD/AlCaDB

Additional code in service - 2 lines to initialise monitoring:

```
from VyPR.init_verification import Verification
verification = Verification(app)
```

Replayed with ~14,500 uploads, observed little overhead, 2 surprising performance deviations.

Subset of the Specification

```
verification_conf = {
  "app.routes" : {
    "check_hashes" : [
      Forall(q = changes('hashes')).\
      Check(lambda q : (
        q.next_call('find_new_hashes').duration() < 0.3
      ))
    ],
    "find_new_hashes" : [
      Forall(c = calls('connection.payload', record=['hash_received'])).\
      Check(lambda c : (
        c.duration() < 0.01
      ))
    ],
   [...]
  },
  [...]
}
```

Determination of Branch Criticality

```
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." % self.tag_in_destination.name)
```

```
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." % self.tag_in_destination.name)
```

In this case, there was a constraint placed over a function call taking place after the conditional.

Using simple scripts written using our analysis library, we determined that the branch taken has no immediate effect on the constraint outcome.

Offline Analysis with VyPR

VyPR provides an object-oriented analysis library that talks to a central server.

Structure of data taken by monitoring is based on the specification, and coupled with the source code.

Examples of complex queries developers can perform include asking:

> Which functions are frequently on paths that cause future specification failure?
>
> Are there common paths between two points that frequently cause slow-down?

Ongoing Work

VyPR will soon be used to monitor the behaviour of the iCMS software.

A new collaborator from University of Lugano will work on making VyPR easier to integrate into Continuous Integration for production software.

VyPR will receive a lot of effort to turn it into an out-of-the-box analysis framework.

VyPR

Thank you for your time!

VyPR is publicly available - cern.ch/vypr

We are looking for new contributors, collaborators and applications:

joshua.dawes@cern.ch