An Analysis Library for VyPR

Progress Report

Joshua Heneage Dawes[1,3,4] Marta Han[2,3]

[1]University of Manchester, UK

[2]University of Zagreb, Croatia

[3]CERN, Geneva, Switzerland

[4]joshua.dawes@cern.ch

Outline

Recap of Analysis-by-Specification

Introduction to Marta's work on the VyPR analysis library

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

An efficient way to do path comparison

Analysis-by-Specification

A performance analysis technique inspired by Runtime Verification (RV).

General idea - specify a property that a part of a program (for us, a function) should hold, and check it at runtime.

Not formal verification; complements testing.

VyPR

Specifications in PyCFTL

Monitoring at runtime for agreement

Offline analysis - used to be done by writing custom scripts, but we're on the way towards having a powerful analysis library.

Analysis Library

Python-based library that connects to a VyPR verdict server for offline analysis

- Standard verdict analysis (was a property violated? was it satisfied?)
- Quantitative verdict analysis (what was the observation that caused failure? how close was it to success?)
- Explanation via State Comparison (was there a valuation of variables that always led to failure? Can we determine clusters in the valuation space?)
- *Explanation via Path Comparison (was there a branch that always led to failure? or was control flow statistically insignificant?)*

Why an Analysis Library?

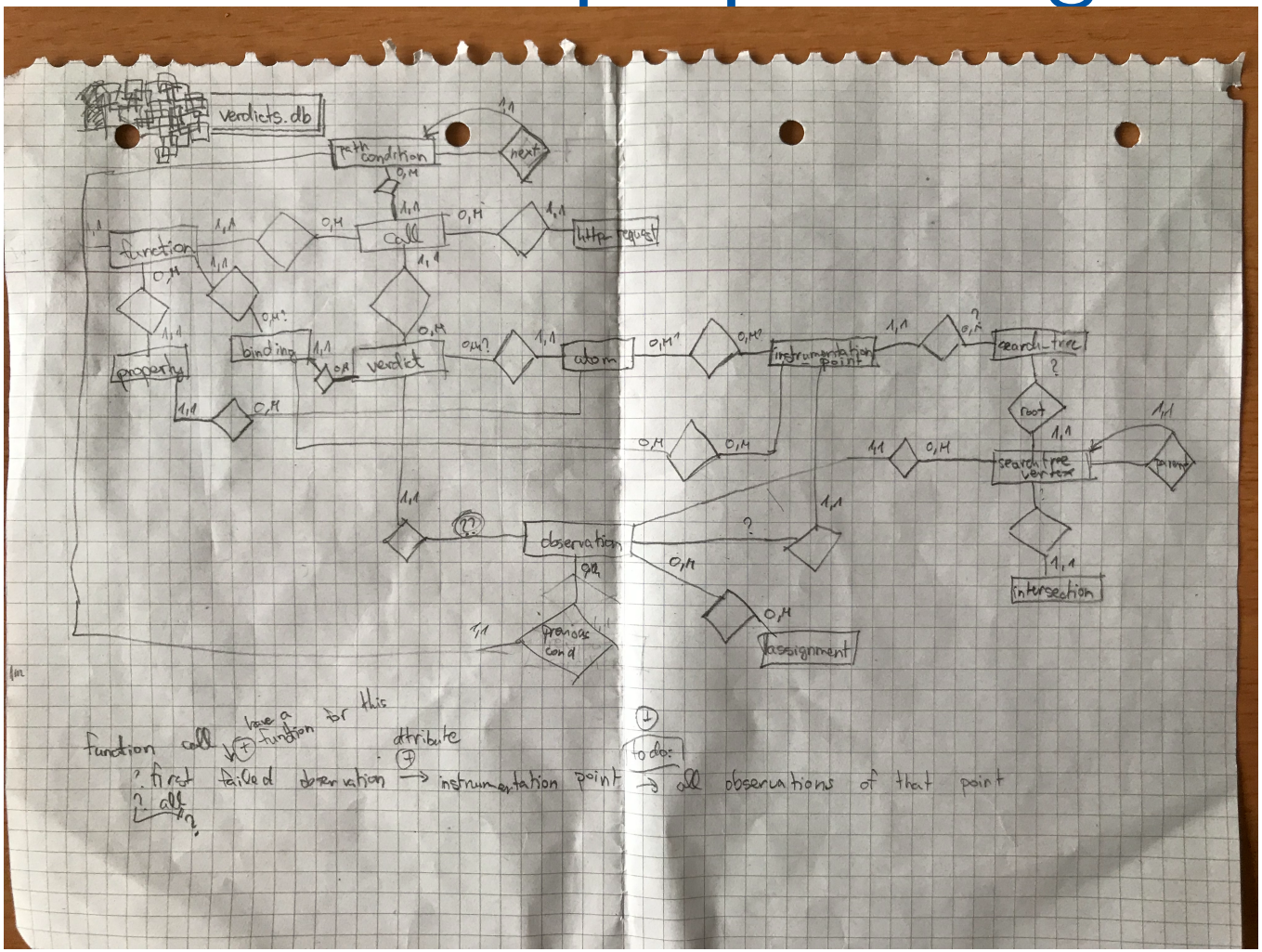
VyPR uses a verdict schema to store the information it extracts during monitoring.

It's gradually becoming more complex - currently 18 tables with 24 foreign keys.

Some exist to deal with extremely unintuitive use cases.

It would be completely unreasonable to leave engineers (or Marta...) at the mercy of that...

We want to avoid people doing this...



What is stored in the database?

HTTP requests - time of request is recorded

Functions - paired with properties

Function calls - which function is called, during which request?

Verdicts - memorise whether the constraint was satisfied and which atom generated this; made during a function call

Observations - store the observed values and resulting truth value; currently paired with program variables' assignments

SQL vs Analysis Library

Finding the time at which a verdict was obtained:

```
select time_obtained
from verdict where id=1;
```

```
v=verdict(1)
time=v.time_obtained
```

Getting all rows in the table that represent function calls during which a failure occurred.

```
select function_call.id,
    function_call.function,
    function_call.time_of_call,
    function_call.http_request
from
function_call inner join verdict
    on verdict.function_call=
        function_call.id
    inner join function on
        function_call.function=function.id
where function.id=1 and verdict.verdict=0
```

```
f=function(1)
calls=f.get_calls_with_verdict(0)
```

A Specification Example

```
verification_conf = {
  "app.routes" : {
    "paths_branching_test" : [
      Forall(
        s = changes('a')
      ).Check(
        lambda s : (
          s.next_call('f', record=['a']).duration()._in([0, 1])
        )
      )
    ]
  }
}
```

The specification places a constraint over calls to f, and we are interested in analysing the 'severity' of violation of that constraint based on recorded variable valuations.

Verdict Severity

Apart from an observation not satisfying the constraint, it might be of interest to know by how much it violates it.

If it does satisfy the constraint, how close is it to violation?

One possible way to measure this is using the following function: the distance between the observed value and the interval boundaries, assigned a negative sign if the value is not in the interval.

$$\mathrm{Sev}(x) = \begin{cases} -\inf_{y \in I} |x - y|, x \notin I \\ +\inf_{y \notin I} |x - y|, x \in I \end{cases}$$

Plot: Verdict Severity vs Time

For a given function, we get a call with failed verdict and find the instrumentation point of the observation responsible for the failure. This means we locate the part of the code which generates an observation which does not satisfy the specification formula.

```
all = f.get_calls_with_verdict(0)[0]
failed_observation = call.first_observation_fail()
inst_point = failed_observation.get_instrumentation_point()
```

We then consider all the observed values (of the measurement constrained by the specification formula, e.g. duration) generated by that part of the code.

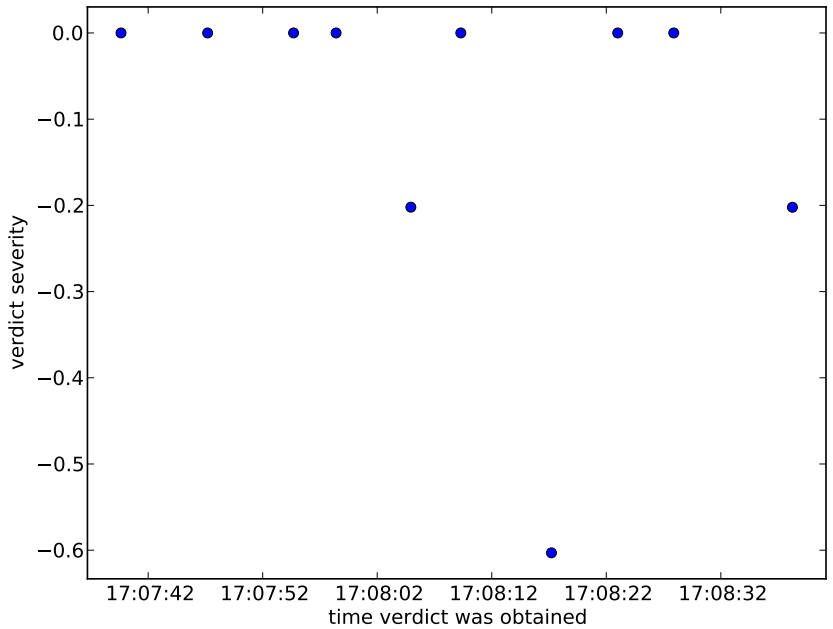
```
observations = inst_point.get_observations()
```

In order to analyse if there are values assigned to the variables at that instrumentation point which are more likely to cause a failure, we group the observations by valuations and create a separate plot for each 'group'.

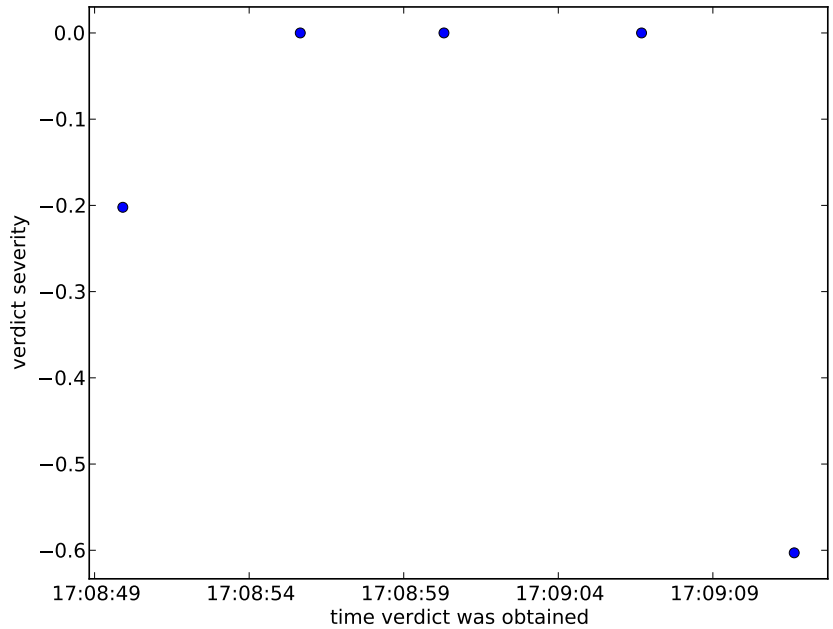
```
valuations=[]
for obs in observations: # ITERATE ON OBSERVATIONS
    assignments=obs.get_assignments()
    print(obs.id)
    final_dict=dict()
    for a in assignments: # CONSTRUCT ASSIGNMENT MAP
        a=vars(a)
        a["value"]=pickle.loads(a["value"])
        print(a)
        final_dict[a["variable"]]=a["value"]

    if final_dict not in valuations:
        valuations.append(final_dict)
        t.append([])
        s.append([])

    # FINALLY, PLOT SEVERITY

    time=analysis.verdict(obs.verdict).time_obtained
    t[valuations.index(final_dict)].append(datetime.strptime(time,
        '%Y-%m-%dT%H:%M:%S.%f'))
    s[valuations.index(final_dict)].append(severity_function(obs))
```

```
s.next_call('f', record=['a']).duration()._in([0, 1])
```

a=10



a=8



We see that these valuations generate both positive and negative severity values. Hence, it is not likely that the assignment values are causing the failure.

Path Comparison

Explanation from another (complementary) angle.

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

Additional instrumentation to record branching

Offline path reconstruction

Comparison via context free grammars

Result? Automated path comparison over time

Recap

We care about reconstructing the path taken by a program up to a given observation.

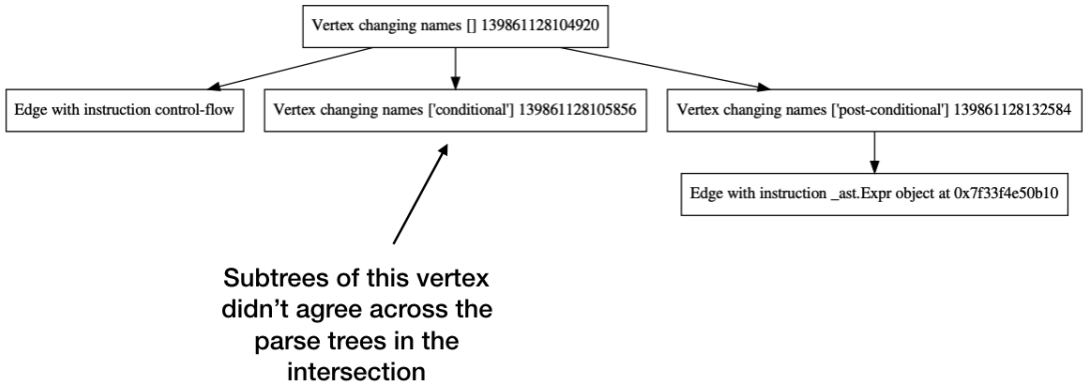
This includes loop multiplicity.

We then ask questions such as "was a particular branch always responsible for a given verdict severity?" or "did a certain number of iterations of a loop cause something to fail?"

Example from a previous talk

And form their intersection

- Computed using an algorithm based on iteratively refining paths through the parse trees.



Some results

Path comparison is expensive, but why?

Steps are

1. Context Free Grammar construction
2. Parse tree derivation
3. Parse tree intersection (this is expensive)

Commutativity as an Optimisation

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

Current implementation of CFG construction, derivation and intersection scales quadratically.

...when we perform intersection from scratch!

If we take advantage of commutativity of intersection, we can store previous results.

A Problem Statement

Path reconstruction is performed by isolating an observation, and finding the path taken (including loop multiplicity).

Path comparison is performed by fixing a set of observations and intersecting the paths up to them.

So, given a set of observations, how do we determine for which subsets there exists an intersection?

Naive Subset Search

Store intersections in a table as a serialised string.

Represent subsets of observations by a table containing subsets.
Many-many relationship between observations and subsets.

Then...

Subset Search continued

Given a subset, we follow the loop:

1. Identify all subsets containing the first element o_1 of our subset (1 query for link table, n queries of subset table for n entries found in link table).
2. Move to the next observation, o_2, in our subset. For each of the n subsets we found in the link table, check if this observation is contained. This requires n queries, and will yield $n - k_2$ remaining subsets (by throwing away k_2 subsets not containing o_2).
3. Continue like this.

So approximately $\Sigma_{i=1}^{n}(n - k_i) = n^2 - \Sigma_{i=1}^{n} k_i \sim n^2$.

Better Search

$O(n^2)$ just for the search stage is not that good.

We can do better (but we have to increase our memory requirement).

BDDs?

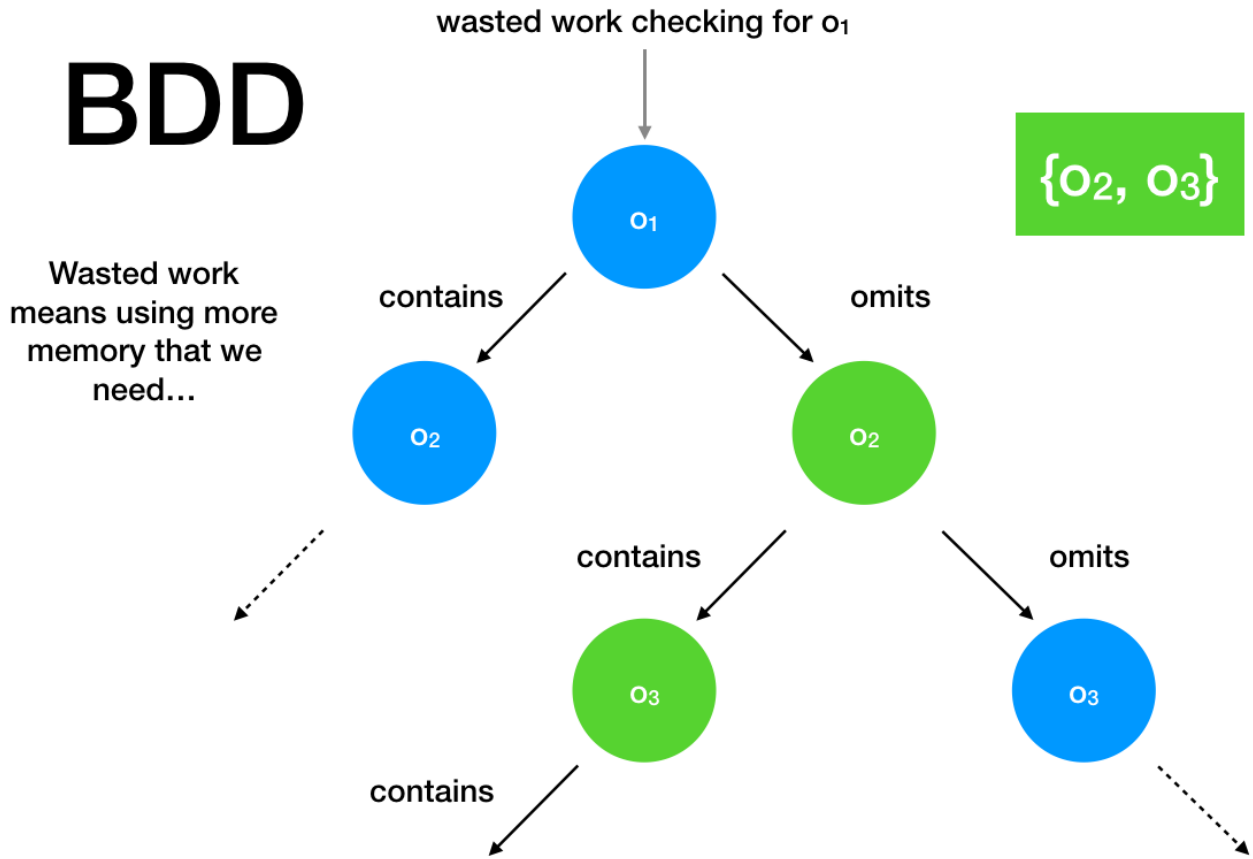
Binary decision diagrams - basically decision trees.

A path through such a tree is a sequence of contains/omits decisions.

But the structure of a BDD is a bit verbose, since each row must be associated with a fixed observation...

So traversal can be a very long process if we have a subset containing observations falling at the top of the tree, and the bottom...

Example


BDD
wasted work checking for o1
{o2, o3}
Wasted work means using more memory that we need…
o1
contains
omits
o2
o2
contains
omits
o3
o3
contains

Search Trees, aka, *set-tries*

A modified decision tree with the properties:

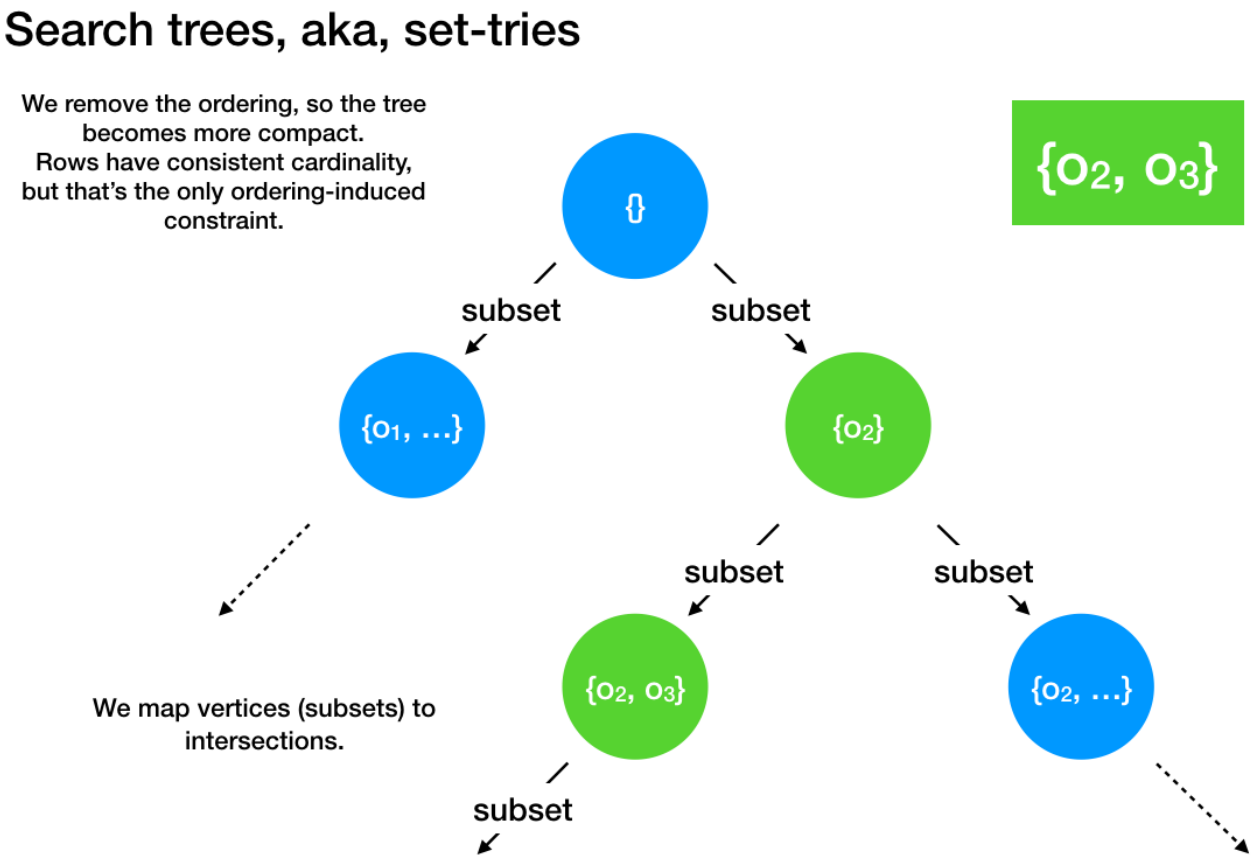
- Non-fixed order on observations.
- In the best case, for n observations, we perform n queries.

How?

Turns out someone had already done something similar, and it's published:

Iztok Savnik. Index Data Structure for Fast Subset and Superset Queries. CD-ARES, Sep 2013. pp.134-148.

Example


Search trees, aka, set-tries
We remove the ordering, so the tree becomes more compact. Rows have consistent cardinality, but that's the only ordering-induced constraint.
{}
{O2, O3}
subset
subset
{O1, …}
{O2}
subset
subset
{O2, O3}
{O2, …}
subset
We map vertices (subsets) to intersections.

A Search Tree Algorithm

1. Check for an existing search tree whose root is the singleton set containing one of the elements of our subset.
    1.1 If there are multiple, take the first we found.
    1.2 If there are none, construct a new search tree with a path through it matching precisely our subset.
2. If we found a search tree, traverse it as far as we can using the elements in our input subset.
3. If we manage to exhaust our set, take the existing intersection.
4. If we arrive at a leaf with remaining observations, we use the intersection there. This is one of the largest existing subsets for which an intersection is already computed. We then extend the tree with the new subset.

A Source of Inefficiency

Suppose we construct a search tree for the set $\{o_1, o_2, o_3\}$.

We then want to compute the intersection of $\{o_1, o_2, o_4\}$. The search tree has a path for some of this, so we only have to intersect one parse tree with that for o_4.

But then we want to compute the intersection for $\{o_2, o_3, o_5\}$.

If we'd set o_2 to be the root, there might be a path giving $\{o_2, o_3\}$, so we could form the intersection with o_5.

Hence, sometimes we can make suboptimal choices because we don't know the future...

Integration into VyPR

The analysis library will send a set of observations (via IDs).

The server's current implementation uses search trees to compute intersection efficiently, giving the result back to the client.

Hence, with the search trees mechanism implemented in the background and an efficient way to return path intersections to the client, the client can easily do performance analysis.

TODO

Path analysis in the client.

Marta is working on an interactive tutorial for VyPR's analysis library with Jupyter notebooks.

The library will soon get its first use with real software.

Initial steps for the tutorial

- Jupyter notebook style.
- We aim for a tutorial based on as realistic an environment as possible.
- This will help new contributors to VyPR, new users and has clear applications in RV teaching.

Import the library

In [ ]:

```
import VyPR_analysis as analysis
```

Setting the URL of the verdict server

In order to change the default server URL, use function set_server with the string containing the new address as an argument.

In [ ]:

```
analysis.set_server("http://localhost:5000")
```

Initialising a class

The name of each class and its attributes match the names of the corresponding table and its columns in the database. In order to initialise an object of class, it is required to pass one or more arguments to the initialising function. In all cases, passing just the ID of the object is enough, as this attribute always determines a row of the table in the database uniquely. However, in some of the tables there are other attributes that belong to a unique row in the table, i.e. the so-called atom index, the name of the function etc. In these cases, it is also possible to initialise the class by passing just one argument. The initialising function then queries the database to get the corresponding row in the table and creates a class with all attributes values. For example, in order to get a variable of class type function, you can use one of the options shown below. The first and the second ones are used to find a row in table function with id=1, whereas the last one determines a row in the table by its name attribute.

In [ ]:

```
f1=analysis.function(1)
f1=analysis.function(id=1)
f1=analysis.function(fully_qualified_name="app.routes.paths_branching_test")
```

Note: in case there are more functions with the same name (which shouldn't happen), the class is initialised as one of the rows without raising an error that there are more of them with the same value.

As some functions in the library, and possibly others defined by the user, might return all the attributes in the row, in order to avoid the inefficiency of querying the database to find what is already known, it is also possible to define the class variable by passing all of the values to the initialising function which, in this case, assigns the given values to corresponding attributes without connecting to the database.

In [ ]:

Code

All of our code is public and found at
`http://github.com/pyvypr/`

. This includes:

- VyPR - the main specification, instrumentation and monitoring machinery for Flask.
- VyPRServer - the central verdict server. Used to be a simple database with some insertion/selection operations, now contains all logic for path reconstruction (not straightforward).
- VyPRAnalysis - Marta's analysis library.
- ContinuousIntegration - scripts for engineers to run local integration tests... coming soon...

Target

CHEP 2019

Analysis Tools for the VyPR Performance Analysis Framework for Python.

J H Dawes, M Han, G Reger, G Franzoni, A Pfeiffer

Paper summarising engineering efforts, with little theoretical content.

The aim is to have results from CMS AICa collaborations (and hopefully DIRAC) to show applications of the new analysis library.

Not a case study paper - a paper about VyPR, with some details on applications to real software.

Questions? :)