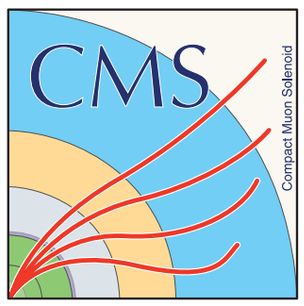
VyPR

cern.ch/vypr

A Framework for Automated Performance Analysis of Python Programs

Joshua Heneage Dawes University of Manchester and CERN
joshua.dawes@cern.ch
Giles Reger University of Manchester
Giovanni Franzoni CERN
Andreas Pfeiffer CERN



Outline

Analysis-by-Specification

A Framework for Analysis-by-Specification

Specification

Instrumentation and Monitoring

A Verdict Server

An Explanation Mode

MANCHESTER 1824
The University of Manchester

VyPR

Analysis-by-Specification

VyPR

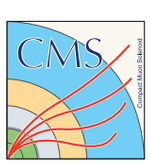
Analysis-by-Specification

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

Runtime Verification - a lightweight formal method.

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.



The University of Manchester

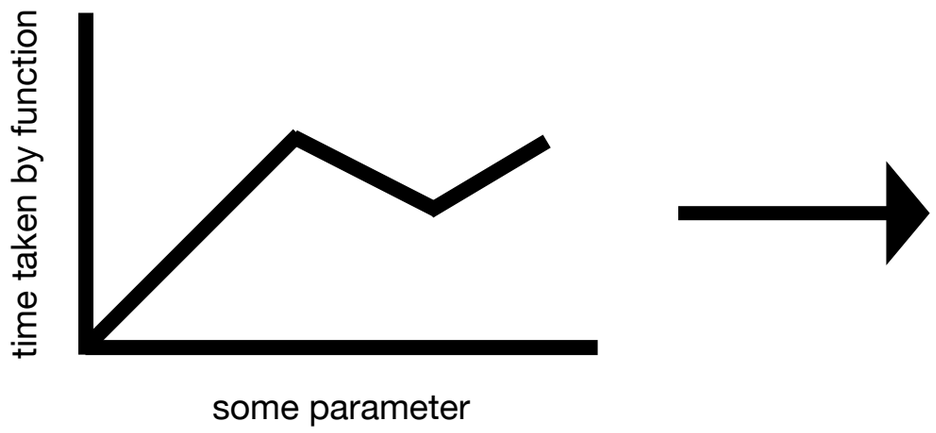
VyPR

Analysis-by-Specification

Runtime Verification for Performance Analysis.

Performance analysis on the CMS Experiment - historically a manual effort.

Analysis using natural language specifications:



When did the execution time of some part of the code exceed some t?

How often?

For which inputs?

CERN CMS MANCHESTER 1824 The University of Manchester

VyPR

Analysis-by-Specification

Express performance requirements in a specification language.

Removes ambiguity inherent in natural language specifications.

We can algorithmically check for satisfaction.

We have precise structure that we can use to explain failure to satisfy specifications.

VyPR

Analysis-by-Specification

the central idea of the VyPR tool

+ some extra machinery to allow sophisticated analysis.

VyPR

VyPR as research output

The first application of Runtime Verification in High Energy Physics (as far as we know) - *J H Dawes, et al. TACAS 19.*

Relatively language agnostic, rigorous theoretical foundation - *J H Dawes, G Reger. SAC 19.*

Research in the environment in which the output will be used.

VyPR

VyPR as a development tool

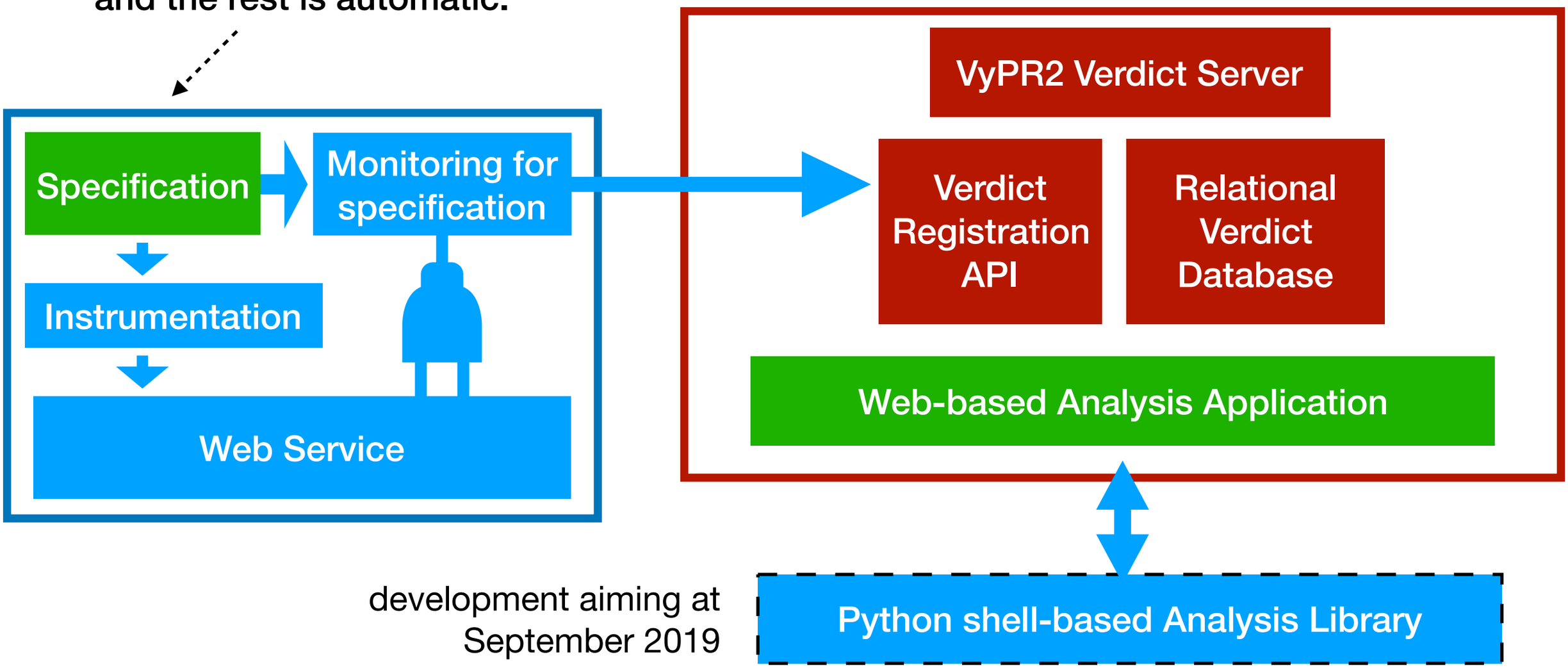
A tool to help developers easily analyse the performance of their Python programs.

No need to know any of the foundation mathematics.

For application to either production systems, or systems still in development.

How does VyPR look?

The developer writes a specification,
adds a line to their service's code,
and the rest is automatic.

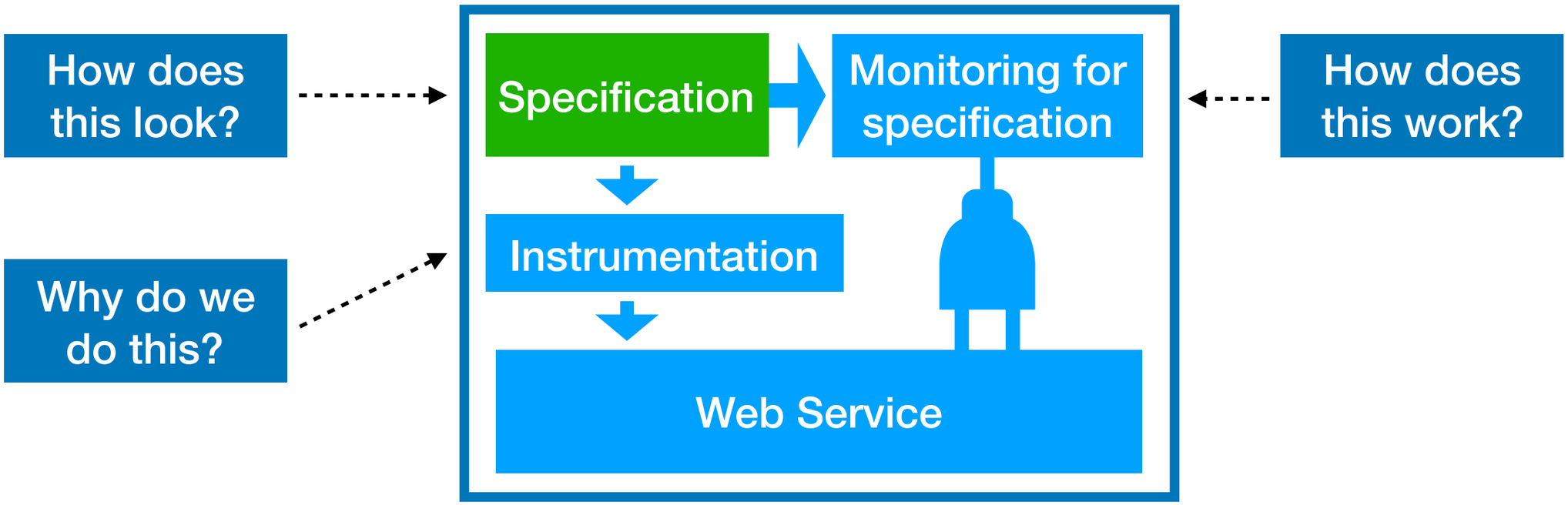


development aiming at
September 2019

CERN CMS MANCHESTER 1824 The University of Manchester

VyPR

Service-level Monitoring



Specification

VyPR provides its own specification language - CFTL.

Control-**F**low **T**emporal **L**ogic

A mathematical logic designed for expressing properties that should hold true over runs of functions in a system.

VyPR provides the PyCFTL library for expressing CFTL specifications.

```
"module description" : {

  "function name" : [

    Forall(q = changes('val')).\
    Forall(t = calls('showData'), after='q').\
    Check(lambda q, t : (
      If(q('val').equals(True)).then(
        t.duration()._in([0, 3])
      )
    ))

  ]

}
```

A full PyCFTL specification.

Let’s investigate…

VyPR

PyCFTL

“Every time val *is assigned, the next call to the function* showData *should take no more than 3 seconds.”*

Selects points of interest at runtime.

```
Forall(q = changes('val')).\
Check(lambda q : (
  q.next_call('showData').duration()._in([0, 3])
))
```

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

VyPR

In more detail

The argument q to the lambda will model *states*, so can be treated as such in the lambda body.

```
Check(lambda q : (
   q.next_call('showData').duration()._in([0, 3])
))
```

Calling *next_call* on a state has the effect of searching forwards in time.

VyPR

More complex…

“Every time val *is assigned, every future call to* showData *should take no more than 3 seconds.”*

Selects points of interest at runtime, using two selections.

One of the selections uses values from the previous as a starting point for the forward search.

```
Forall(q = changes('val')).\
Forall(t = calls('showData'), after='q').\
Check(lambda q, t : (
  t.duration()._in([0, 3])
))
```

VyPR

In more detail

The argument q to the lambda will model *states*, and t will model function calls.

```
Check(lambda q, t : (
  t.duration(). in([0, 3])
) )
```

Once we've measured duration, we can define a predicate over it.

Naturally, we can measure the duration of the function call represented by t.

Checking for branches

What if we only want to place a constraint on a branch whose entry condition is that *val* is True?

```
Forall(q = changes('val')).\
Forall(t = calls('showData'), after='q').\
Check(lambda q, t : (
  If(q('val').equals(True)).then(
    t.duration()._in([0, 3])
  )
))
```

We only care about the time taken by calls to *showData* if *val* is equal to True.

VyPR

A complete specification format

```
"module description" : {

  "function name" : [

    Forall(q = changes('val')).\
    Forall(t = calls('showData'), after='q').\
    Check(lambda q, t : (
      If(q('val').equals(True)).then(
        t.duration()._in([0, 3])
      )
    ))

  ]

}
```

Separation of module and function names makes resolution easier.

No modification of source code.

Specification for entire service written in one file.

VyPR

Instrumentation and Monitoring

VyPR

```
Forall(q = changes('val')).\
Forall(t = calls('showData'), after='q').\
Check(lambda q, t : (
  If(q('val').equals(True)).then(
    t.duration()._in([0, 3])
  )
))
```

How do we guarantee that we will receive enough information from the program to check this, while preserving program behaviour?

VyPR

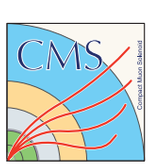
Trace functions?

The Python interpreter gives us the nice option of a trace function.

Every event generated at runtime by the interpreter triggers a call to the trace function.

For every event, we make a decision on whether we need it.

Potentially massive overhead.



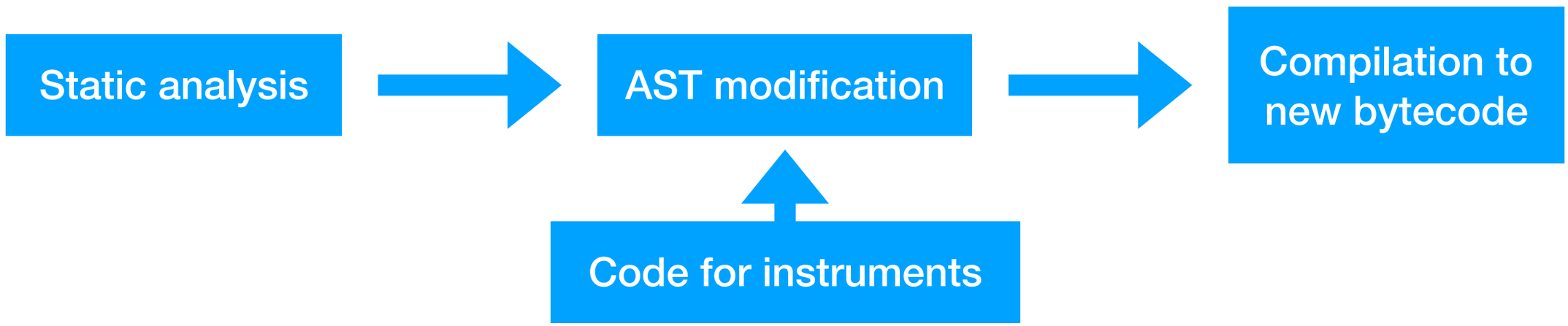
VyPR

The need for instrumentation

If a trace function (implemented in either Python or C/C++) is not an option, what do we do?

VyPR uses static analysis to perform “instrumentation”.

“Instrumentation” - adding code to the program so we can check specifications.



Monitoring

Given data from instruments, how do we check if a specification holds?

CFTL's monitoring algorithm is not straightforward… but it's efficient.

VyPR sets up a separate thread, to which instruments send their data via an intermediate consumption queue.

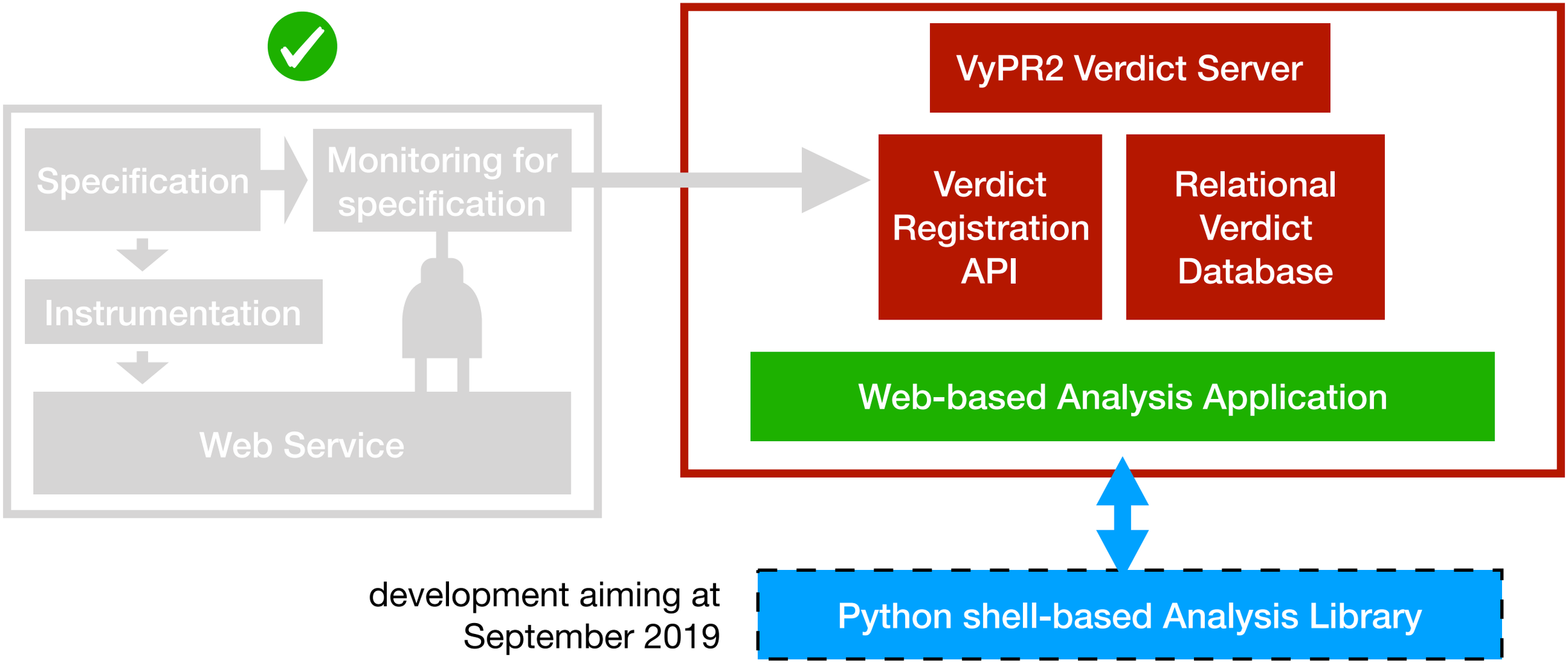
VyPR

Monitoring

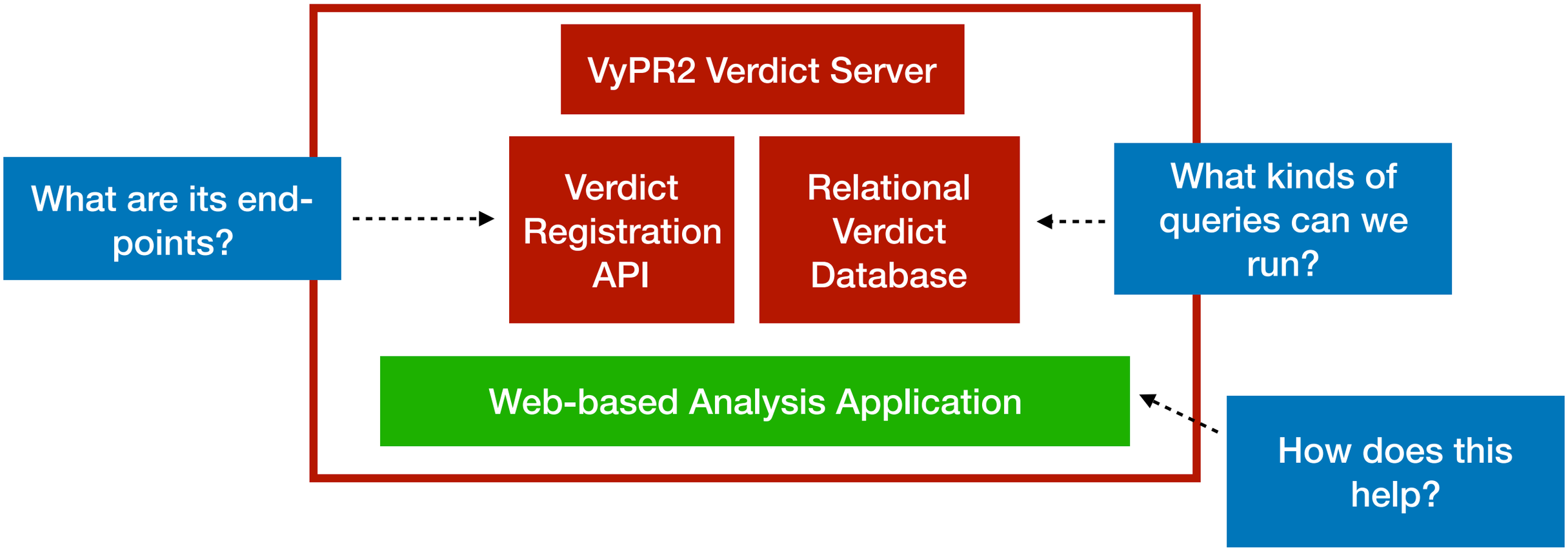
So far, CFTL specifications are written over single function calls, hence single threads.

This means we send the relevant data to the verdict server once a function's execution has ended.

We almost have a complete architecture…



development aiming at September 2019


VyPR2 Verdict Server
What are its end-points?
Verdict Registration API
Relational Verdict Database
What kinds of queries can we run?
Web-based Analysis Application
How does this help?

A Verdict Database

1) The top level is *transactions*.

2) All verdict data is grouped by the function/property pair for which it was obtained, then by calls to that function.

```
"module description" : {

  "function name" : [

    Forall(q = changes('val')).\
    Forall(t = calls('showData'), after='q').\
    Check(lambda q, t : (
      If(q('val').equals(True)).then(
        t.duration()._in([0, 3])
      )
    ))

  ]

}
```

VyPR

A Verdict Database

```
"module description" : {

  "function name" : [

    Forall(q = changes('val')).\
    Forall(t = calls('showData'), after='q').\
    Check(lambda q, t : (
      If(q('val').equals(True)).then(
        t.duration()._in([0, 3])
      )
    ))

  ]

}
```

◄-------- 3) We then group by points of interest.

VyPR

A Verdict Database

```
"module description" : {
  "function name" : [
    Forall(q = changes('val')).\
    Forall(t = calls('showData'), after='q').\
    Check(lambda q, t : (
      If(q('val').equals(True)).then(
        t.duration()._in([0, 3])
      )
    ))
  ]
}
```

4) We then identify the part of the property that caused collapse to a verdict.

VyPR

A Web Application for Simple Analysis

VyPR

Verdict Data

Not Secure | wvypr-vm.cern.ch:9001

Flask-VyPR Analysis Tool Search by Specification Search by Verdict

VyPR, CFTL and PyCFTL

Flask-VyPR is a tool for runtime verification of Python and Flask-based web services with respect to specifications written in Control Flow Temporal Logic.

It consists of:

- The PyCFTL library, a small library for writing CFTL specifications in Python.
- A program that performs optimal instrumentation of the web service for the given CFTL specifications.
- A library for *making the web service verified* at runtime.
- A server for collection of verdict data from the web service's runtime. This server provides both an API for a service under scrutiny to store its verdict data, and a front-end for users to either 1) see verdicts from certain parts of the service or 2) see which parts of the service gave a specific verdict.

Control Flow Temporal Logic (CFTL) is a specification language in the form of a temporal logic used to specify constraints over the state and time of a program.

PyCFTL is the Python binding for CFTL that allows CFTL properties to be written in Python and checked over a Python program.

This tool shows verdict data from CFTL formulas written in PyCFTL to describe behaviour of web services.

Starting from the function/property pair, you can see the verdicts reached by verifying the property over the function in a specific HTTP request.

Search by Specification

Starting from a verdict, find the code that generated that verdict at some point during runtime.

Search by Verdict

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	Verdict
app	Lines [74] **Violation**
routes	reached at 2019-05-24T12:19:49.539417
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 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

Selection of function, property, transaction and call.

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

Verdicts are listed with their line numbers and timestamps.

Verdict Data

Not Secure | wvypr-vm.cern.ch:9001/verdict/

Flask-VyPR Analysis Tool | Search by Specification | Search by Verdict

CERN | 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

Verdict Data

Not Secure | wvypr-vm.cern.ch:9001/verdict/

Flask-VyPR Analysis Tool | Search by Specification | Search by Verdict

CERN | 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

Selection of verdict and subsystem.

Verdict Data

Not Secure | wvypr-vm.cern.ch:9001/verdict/

Flask-VyPR Analysis Tool | Search by Specification | Search by Verdict

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

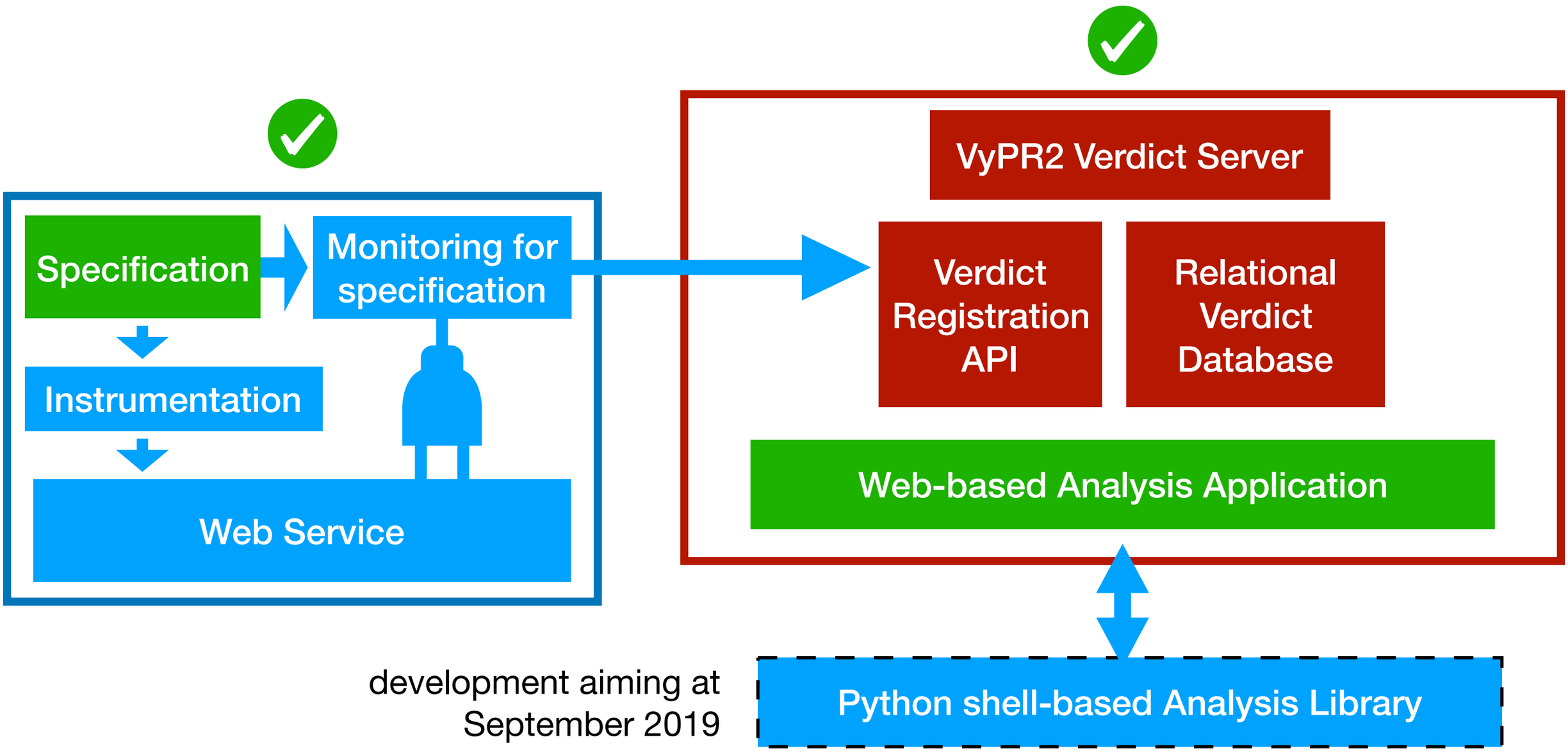
app.routes.paths_branching_test

```
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

Relevant verdicts are listed.

Architecture for Analysis-by-Specification? Check.



development aiming at September 2019

How has this already helped?

VyPR has profited a lot from being developed at CERN.

The first major application - the next version of CMS' non-event data upload service.

Worked closely with CMS' Alignment and Calibrations group - thanks to Giacomo Govi.

During replay of 6 months of uploads from LHC runs, VyPR found unexpected performance drops, while inducing little overhead. *J H Dawes, et al. TACAS 2019.*

VyPR

How has this already helped?

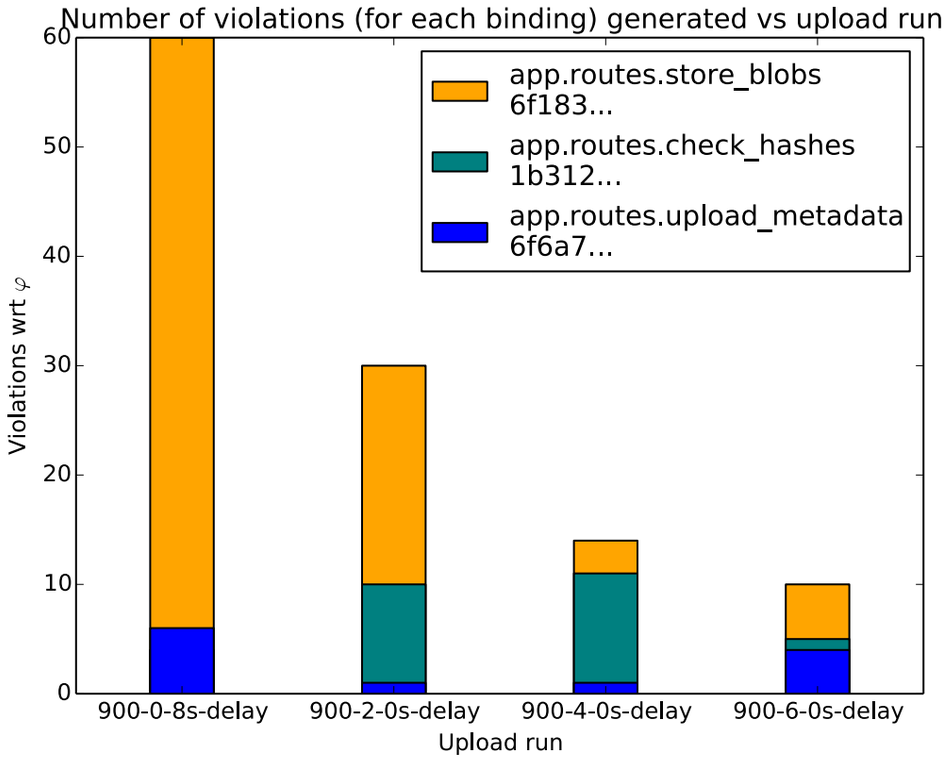
A bit of context

“CMS non-event data uploads” - during LHC runs, non-event data are uploaded to a central database ready for reconstruction.

VyPR was used to analyse the service used for such uploads.

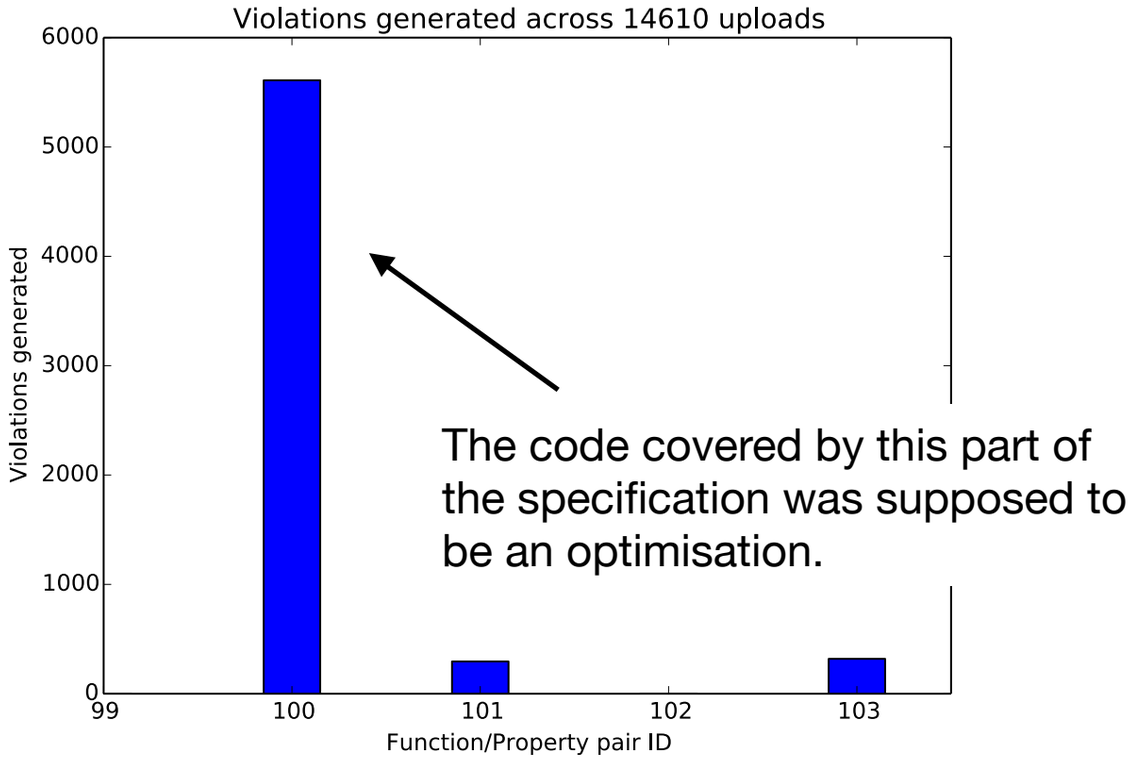
VyPR

How has this already helped?



Y axis is number of violations

X axis is how long we waited between uploads



Y axis is number of violations

X axis is pairs of functions with properties they should satisfy

CERN CMS MANCHESTER 1824 The University of Manchester

VyPR

Lessons learned?

The first analysis tools we built were not powerful enough.

Analyses performed on non-event upload required custom scripts for verdict database querying.

We need an automated explanation mechanism.

How could an explanation look? What would be useful to developers?

An Explanation Mode for VyPR

Which parts of code led to failing observations more frequently?

VyPR

```
if n > 10:
    l = []
    for i in range(n):
        l.append(i**2)
else:
    l = []
showData(l)
return True
```

```
"module" : {

  "function" : [

    Forall(t = calls('showData')).\
    Check(lambda t : (
      t.duration()._in([0, 3])
    ))

  ]

}
```

Suppose the verdict database tells us that there was a call to *showData* that failed this constraint.

Knowing the paths through code taken to reach the failing observation across multiple runs might indicate “faulty control flow”.

Distinction between paths => the difference found may correlate to performance drops.

No distinction => control flow is *probably* not responsible.

VyPR

```
if n > 10:
    l = []
    for i in range(n):
        l.append(i**2)
else:
    l = []
showData(l)
return True
```

```
"module" : {

  "function" : [

    Forall(t = calls('showData')).\
    Check(lambda t : (
      t.duration()._in([0, 3])
    ))

  ]

}
```

Suppose the verdict database tells us that there was a call to *showData* that failed this constraint.

Knowing the paths through code taken to reach the failing observation across multiple runs might indicate “faulty control flow”.

Distinction between paths => the difference found may correlate to performance drops.

No distinction => control flow is *probably* not responsible.

VyPR

```
if n > 10:
    l = []
    for i in range(n):
        l.append(i**2)
else:
    l = []
showData(l)
return True
```

```
"module" : {

  "function" : [

    Forall(t = calls('showData')).\
    Check(lambda t : (
      t.duration()._in([0, 3])
    ))

  ]

}
```

Suppose the verdict database tells us that there was a call to *showData* that failed this constraint.

Knowing the paths through code taken to reach the failing observation across multiple runs might indicate “faulty control flow”.

Distinction between paths => the difference found may correlate to performance drops.

No distinction => control flow is *probably* not responsible.

VyPR

Integration of Path Reconstruction

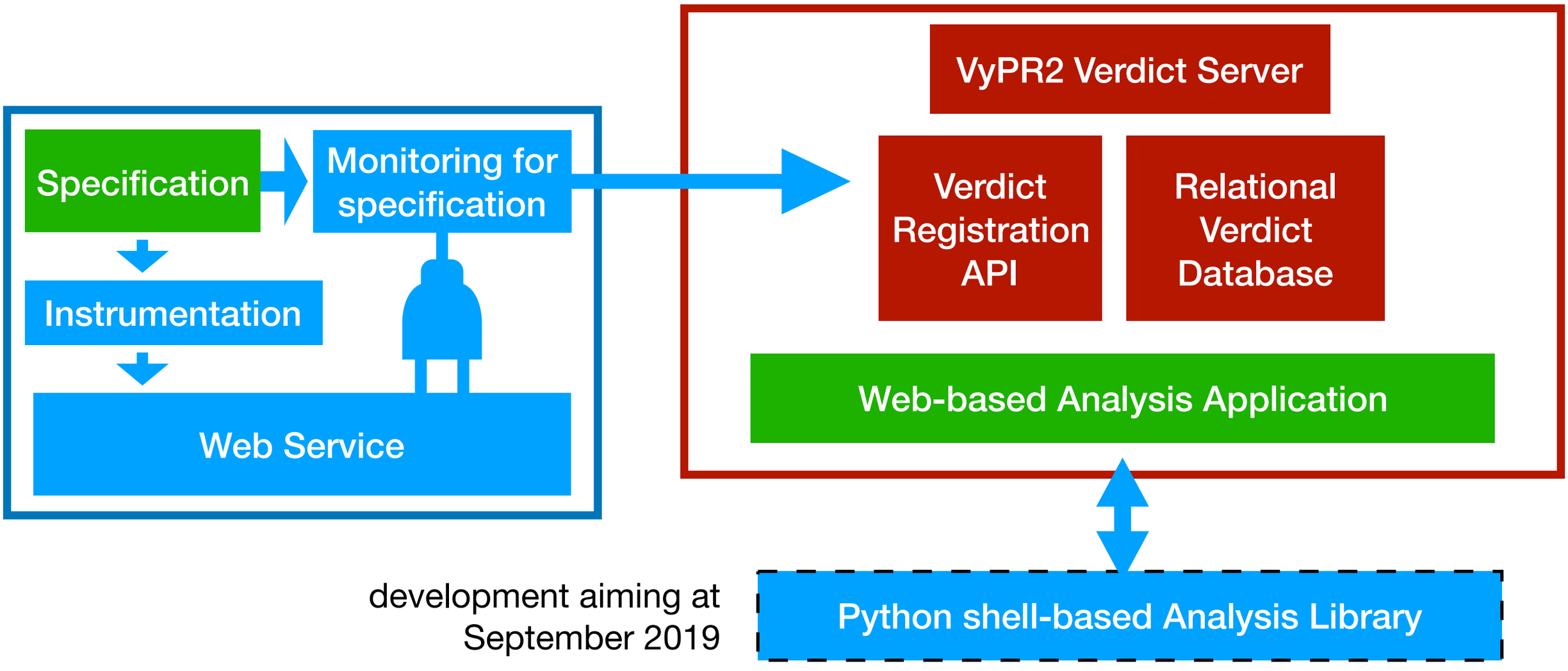
VyPR's architecture integrates the machinery for path reconstruction efficiently in space and time.

We store the minimum data needed to be able to reconstruct paths in most cases.

Path Reconstruction becomes more efficient as we observe more from the monitored system.

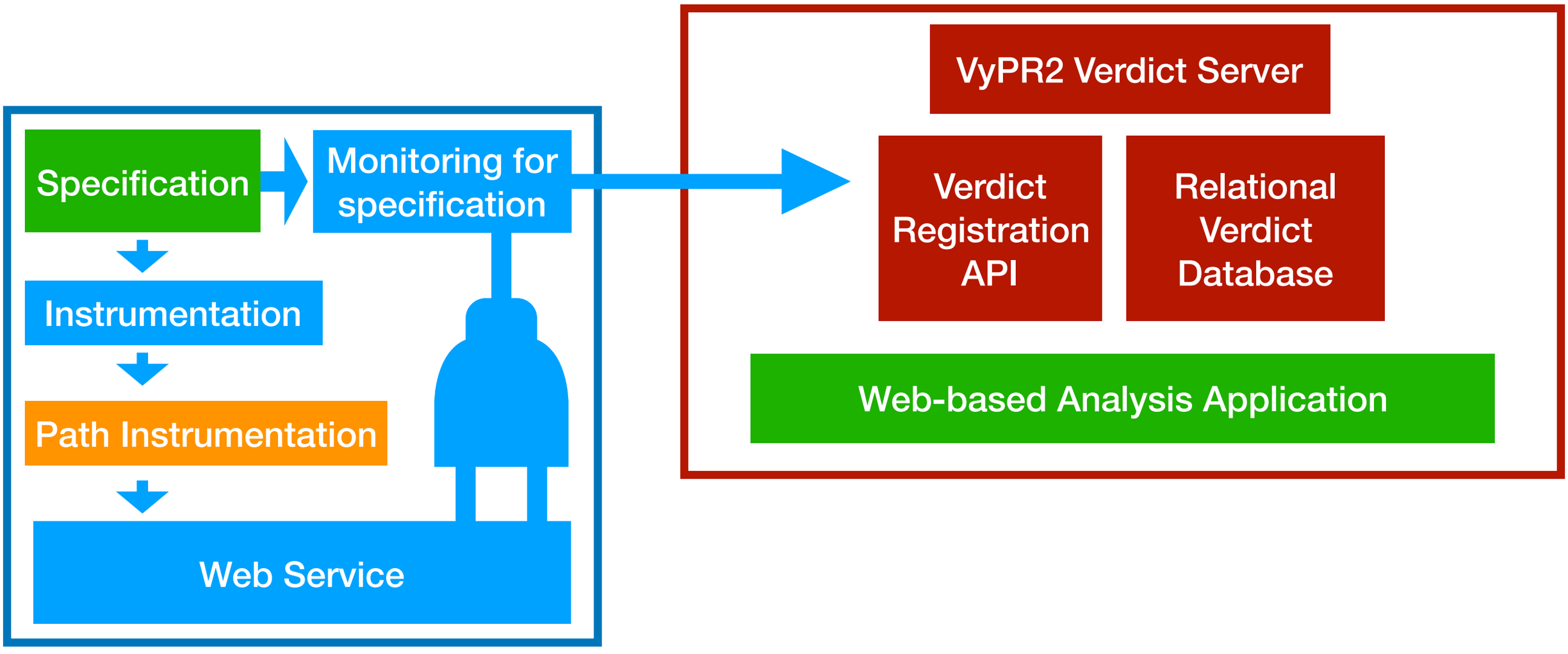
VyPR

Path Comparison

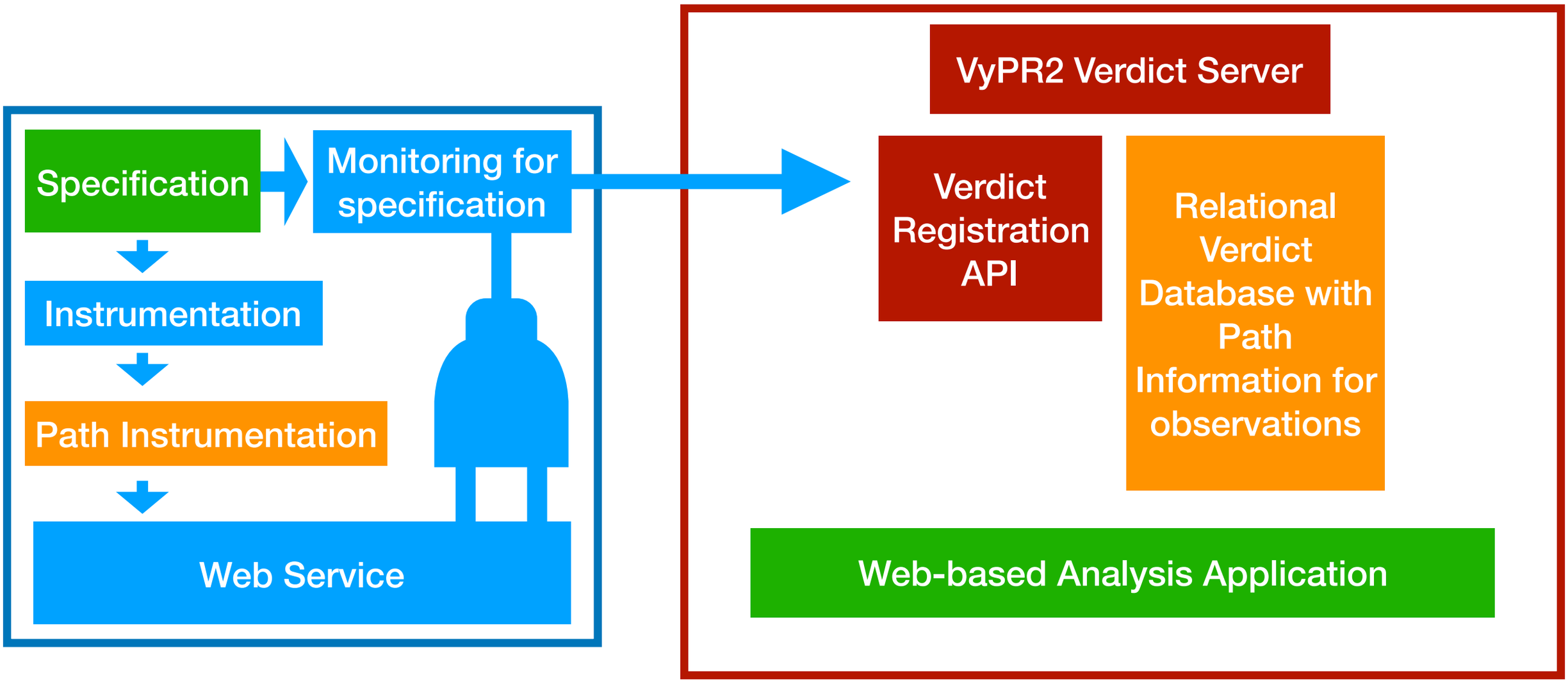


development aiming at September 2019

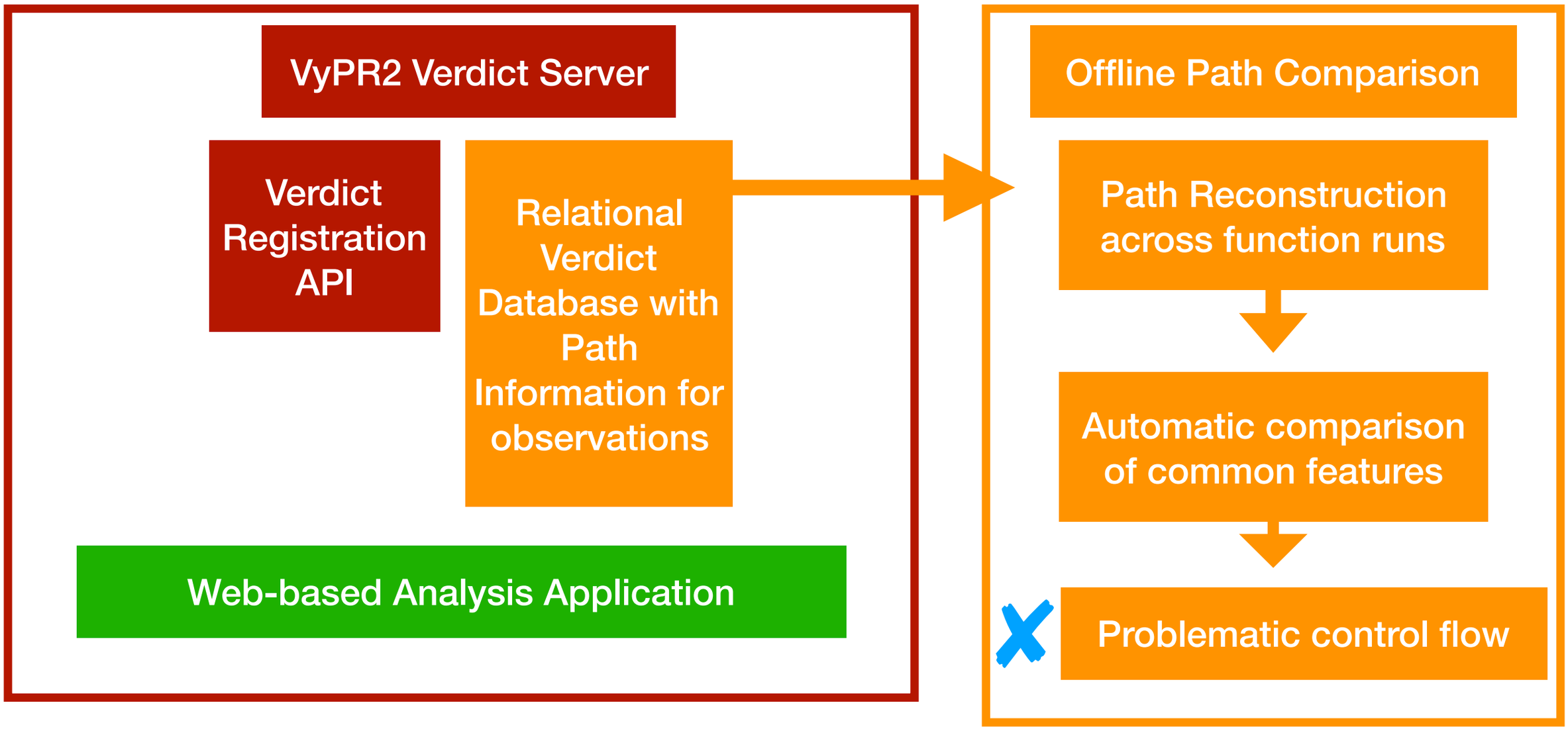
Path Comparison



Path Comparison



Path Comparison



CERN CMS MANCHESTER 1824 The University of Manchester

VyPR

Common Features?

Given two reconstructed paths that are similar except for some branches taken, VyPR can detect the locations of the deviations.

```
if n > 10:
    l = []
    for i in range(n):
        l.append(i**2)
else:
    l = []
showData(l)
return True
```

```
if n > 10:
    l = []
    for i in range(n):
        l.append(i**2)
else:
    l = []
showData(l)
return True
```

VyPR can determine that the conditional is the source of deviation.

```
if n > 10:
    l = []
    for i in range(n):
        l.append(i**2)
else:
    l = []
showData(l)
return True
```

VyPR

Plans for an Analysis Library

Next 6 months - development of a powerful analysis library.

Plans are for this to be a self-contained library for use in the Python shell or as CLI.

Make explanation an intuitive process for developers.

VyPR

Conclusions

VyPR is a research tool developed in an industrial environment, and has already proven its utility.

Still more design decisions to be made.

Theory is in a good place - but the analysis tools need to be developed - will happen over the next 6 months.

VyPR

During the next year of work on VyPR and inline with the so-far use-case driven development of theory and implementation:

We are looking for use cases around CERN, and further afield.

cern.ch/vypr - joshua.dawes@cern.ch

Thank you for listening!

VyPR