wVyPR: A Step Towards Automated Performance Analysis of Web Services

Control-Flow Temporal Logic, VyPR and wVyPR

Joshua Dawes [1,2] Giles Reger [1] Giovanni Franzoni [2]
Andreas Pfeiffer [2]

[1]University of Manchester, Manchester, UK

[2]CERN, Geneva, Switzerland

Contents of this talk

Performance analysis by specification

We think of "performance analysis" as the activity of profiling things like function calls and looking at the resulting plots.

Why build plots? To look for unusual behaviour or patterns, ie, to check properties given by some specification.

When we analyse performance manually, the specification is in natural language - "some function should always take no more than 10 seconds to run".

Performance analysis by specification involves expressing the checks we would do manually in a formal specification.

Runtime verification looks at performing the checks for us.

Why formal specifications?

Given a specification in natural language, having a generalised way to check this isn't really feasible...

Instead we design a precisely defined language - a logic.

A logic is a language we use to write formulas to express a demand that something must be true.

Formulas over runs of programs deal with truth that varies over time.

Result: Formal specifications are usually written in Temporal Logics.

Also: if a property is written down formally, finding a general way to explain violations is much more feasible than with specifications in natural language.

CFTL: a specification language

We abstract (single-threaded) program runs into sequences of states and transitions between them.

States are maps from program symbols (variables, functions) to values.

Of states, we can measure values (to which the state maps some symbol).

Transitions are pairs of states.

Of transitions, we can measure durations (the time taken to move between states).

We formally refer to this system as a Dynamic Run (DR).

CFTL continued

CFTL reasons over single runs of functions, modelled by single DRs.

It can be used to write formulas that describe performance by:

1. Writing about durations of transitions and values to which states map symbols.
2. Linking what we write via propositional connectives ($\vee$, $\neg$, etc).

A central idea in CFTL is to take properties formed like this and apply them to points of interest by quantification.

eg, "for every change to x, if $x < 5$ then the next call to f should take no more than 10 seconds.".

CFTL by example

"For every change to x, if $x < 5$ then the next call to f should take no more than 10 seconds."

First, we write the inner part. Suppose q models states changing x:

$$q(x) < 5 \implies \text{duration}(\underbrace{\text{next}(q, \text{calls}(f))}_{\text{next call to } f \text{ after } q}) < 10$$

Then, we say that this should be true whenever x is changed:

$$\forall q \in \overbrace{\text{changes}(x)}^{\text{every state changing } x\text{, bound to } q \text{ one-by-one}} : \quad q(x) < 5 \implies \text{duration}(\text{next}(q, \text{calls}(f))) < 10$$

Verifying a Program P wrt a Property φ

Once we have a specification language, for a property φ, we must:

Instrument P minimally so we observe everything needed to monitor for φ.

Monitor at runtime to reach a verdict.

VyPR

A complete runtime verification tool for CFTL properties written about single-threaded Python programs.

Provides PyCFTL, a library for writing CFTL in Python.

Automatically instruments the input program minimally.

Monitors efficiently at runtime.

Generates a verdict report - "Which parts of my program caused a violation?"

A Departure from Conventional RV Approaches

VyPR focuses on specification from the engineer's perspective.

Instrumentation is a key part of verification, rather than an optimisation.

Existing approaches see instrumentation as a way to reduce what we observe at runtime, starting from everything... surely it's more intuitive to see instrumentation as something we have to solve for a specification language to be useful?

With instrumentation as a key part of verification, monitoring becomes significantly more efficient.

Thinking about moving to web services

Starting from VyPR, we have minimal instrumentation and efficient monitoring for free.

To move to web services, we have to address some problems.

PROBLEM 1

VyPR's monitoring algorithm requires instrumentation information - this is held in memory.

We should avoid instrumentation whenever a service starts (if the code hasn't changed) - so we have to do it in a separate process.

...meaning instrumentation data has to be persistent between the instrumentation process and the actual service.

Thinking about moving to web services

PROBLEM 2

VyPR verifies a single function wrt a single property.
Web services consist of multiple functions, and we should be able to verify each with respect to multiple properties.

Solutions

PROBLEM 1 Use Python's `pickle` module to dump instrumentation data to files. When a service starts, the verification library reads in the serialised instrumentation data from these files and makes it available to the monitoring algorithm.

PROBLEM 2 Adjust instrumentation so that instruments tell the monitoring algorithm 1) in which function they are placed and 2) for which property they are placed.
After extending VyPR based on these solutions, we get the wVyPR framework.

wVyPR

Service-level instrumentation and monitoring, combined with a separate server for analysis of verdicts obtained by monitoring specifications.

A CFTL specification for the whole service is written in a central file, `verification_conf.py`.

Instrumentation modifies the service source code (inserts instruments) wrt the specification.

Online monitoring of the specification works with very little overhead (experiments show $\approx 5\%$, with a minimum observed so far at $\approx 1\%$).

The verdict server provides querying either by function (to find a list of verdicts) or by verdict (to find a list of generating functions).

Monitoring a Service

Write your PyCFTL specification in `verification_conf.py`.

Given a Flask application object app, add the code

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

Once wVyPR is developed further, this will be all you need to do.

```
"app.usage" : { # module
  "Usage.new_upload_session" : [ # function/method
    Forall( # quantification
      StaticState('q', 'authenticated')
    ).Forall(
      StaticTransition(
        't',
        'self.connection.execute', uses='q'
      )
    ).Formula( # formula to apply
      lambda q, t : (
        If(
          q('authenticated').equals(True)
        ).then(
          t.duration()._in([0, 1])
        )
      )
    )
  ]
}
```

$$\forall q \in \text{changes}(\texttt{authenticated}) :$$
$$\forall t \in \text{future}(q, \text{calls}(\texttt{self.connection.execute})) :$$
$$q(\texttt{authenticated}) = \texttt{True} \implies \text{duration}(t) \in [0, 1]$$

"Whenever `authenticated` is changed, if it's set to `True`, then every future call to `self.connection.execute` should take no more than 1 second."

Conditions Uploader

Our first test case is the Conditions Uploader.

A single-threaded, Python-based web service with plenty of test data (upload replays recorded over 6 months).

The approach taken has been to cover the upload process by a specification.

We can use wVyPR's verdict server to do offline analysis of verdicts.

Generating Verdicts

For each run of an upload dataset (a set of uploads), we get a snapshot of the verdict database.

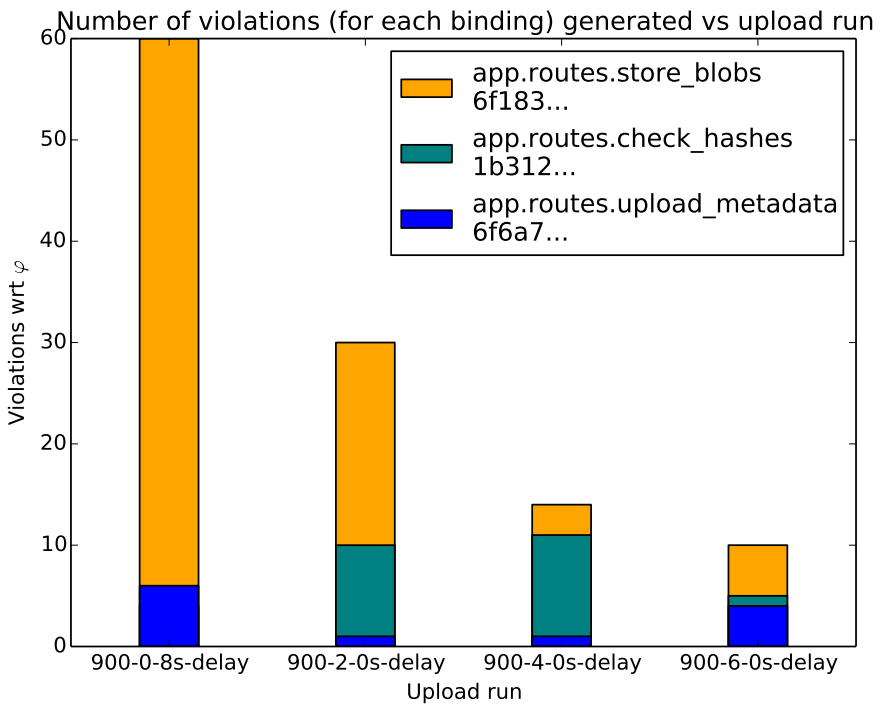
This snapshot encodes which functions calls generated which verdicts wrt the specification defined on the service level.

By querying a single snapshot, or collections of such snapshots, we get useful insights into performance.

Runtime Verification is case-specific

We write a specification, but the analysis of the verdicts generated by the specification during runtime depends on the application.

This plot shows verdicts generated by a 900 upload dataset (which uploaded ≈ 300 payloads) repeated with varied delays between uploads - the plot was generated by processing a sequence of verdict database snapshots.

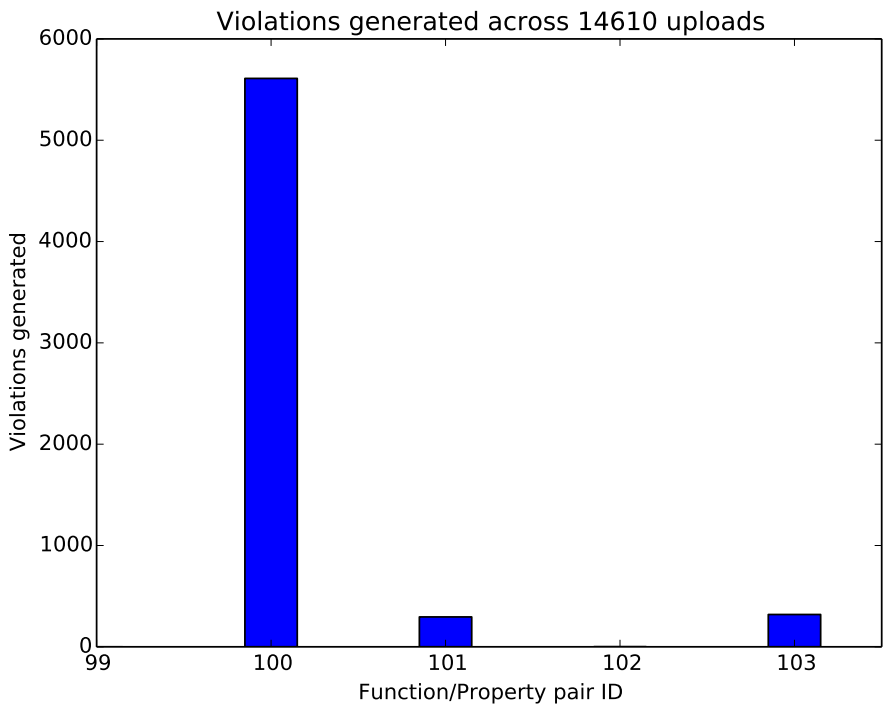


Orange is violations of a constraint on payload uploads.
Curiously, violations of that constraint are more frequent when we decrease the delay.

(Almost) Complete Conditions Replay

x is the ID of the function/property pair being monitored. $x = 100$ refers to a constraint on hash checking.

$$\forall q \in \text{changes}(\texttt{hashes}) :$$
$$\text{duration}(\text{next}(q, \text{calls}(\texttt{check\_hashes}))) \leq 0.3$$



Improving the Verdict Server

The verdict server allows one to find 1) which verdicts were generated by a particular section of the source code of a service; and 2) which sections of a service generated a particular verdict.

Analysis done during the Conditions uploader study has shown that the current verdict server does not provide enough tools to do meaningful analyses.

I intend to spend some time working on improving wVyPR's verdict analysis tools.

The current approach is based on using a web-based GUI - I think a library (like CondDBFW for Conditions...) would be useful.

Publication

We will submit a tool paper in November.

The paper is a mirror of this talk.

wVyPR will be released to the public as a single tool for verification and offline analysis.

Next Step

To have automate performance analysis, we need to be able to explain what went wrong...

"If this property failed somewhere, why?" - depending on the way in which a property was violated, the approach varies.
Suppose Γ is a set of states or transitions over which we quantify. Then:

$$\begin{aligned}(\forall q \in \Gamma : \psi(q)) \equiv \bot &\iff \neg(\forall q \in \Gamma : \psi(q)) \equiv \top \\ &\iff (\exists q \in \Gamma : \neg\psi(q)) \equiv \top\end{aligned}$$

For $q \in \Gamma$ such that $\neg\psi(q)$ holds, why does it hold? If we observe a sequence of states/transitions, which one moved $\psi(q)$ from ? to $\bot$?

Static Slicing - a *local* approach

Given some $q \in \Gamma$ such that $\neg\psi(q)$ holds, this is the case because we observed a state or transition that failed our property.

If we observed a transition, suppose it's a function call and has parameters $\{p_i\}$.

If we observed a state, then there are a set of symbols $\{s_i\}$ that caused the violation.

Either way, we can isolate a set of variables $\{v_i\}$ (from either p_i or s_i) and construct, statically, a backwards dependency graph - a graph whose vertices model instructions and edges model dependence between them.

Traversing a graph like this yields a set of parameters $d_1, \ldots, d_n$ which contribute to the truth value of $\psi(q)$.

Explanation?

In the static slicing approach, explanation is the construction of classes of the n-dimensional space D generated by finding the set of dependence parameters $d_1, \ldots, d_n$.

Some family of subspaces $D'_1, \ldots, D'_k \subseteq D$ may be known to always yield $\psi(q) \equiv \perp$.

Note that I say classes and not partitions because I want to emphasise the fact that it may not be the case that one can construct disjoint classes - especially if we have non-determinism!

Any questions?