Towards Automated Performance Analysis of Python Programs

University of Manchester Formal Methods Seminar

Joshua Heneage Dawes[1,2,3] Giles Reger[1] Giovanni Franzoni[2] Andreas Pfeiffer[2]

[1]University of Manchester, Manchester, UK

[2]CERN, Geneva, Switzerland

[3]`joshua.dawes@cern.ch`, `http://cern.ch/jdawes`

- I'm a Doctoral Student based at CERN, with Manchester as home institute.
- In this seminar, I will describe what is, to the best of our knowledge, the first application of Runtime Verification in High Energy Physics and to web services.

Context

- The work in this seminar is described across 3 papers:
    - Specification of State and Time Constraints for Runtime Verification of Functions `https://arxiv.org/abs/1806.02621`
    - Specification of Temporal Properties of Functions for Runtime Verification *to appear in SAC 2019*
    - VYPR2: A Framework for Runtime Verification of Python Web Services *to appear in TACAS 2019*
- More information about the result of this research can be found at `http://cern.ch/vypr`.

Runtime Verification: *A Classical View*

We wish to check, at runtime, whether some program P holds a property φ written in some temporal logic, for example Linear-time Temporal Logic or Metric Temporal Logic.

- A monitor is synthesised for φ.
- Such a monitor is often an automaton $\mathcal{A}_\varphi$.
- Runs of P are abstracted into *traces* τ, holding enough information to check φ.

Practicalities

- Typically, work on Runtime Verification focuses on a setting where a trace τ has *already been derived* from a run of a program P.
- Further, specifications are often high-level.
- What does the LTL formula $\mathcal{G}(p \to \mathcal{X}(q))$ actually mean when applied to a program? We need an *instrumentation mapping*.

$$\begin{aligned} p &\leftrightarrow \texttt{x} < 10 \\ q &\leftrightarrow \text{call } \texttt{function} \end{aligned}$$

RV for Performance Analysis

- Performance Analysis performed at CERN normally consists of profiling a system and looking at plots.
- The purpose of deriving plots is normally to check them for some property in one's head expressed in natural language.

RV for Performance Analysis

- What if we could encode performance requirements as formulas in a logic and apply RV?
- Then we could consistently synthesise checking mechanisms for performance requirements.
- Maybe then explanation could be automated to some degree...
- While doing all of this, we need a specification language that's accessible to engineers.

Control-Flow Temporal Logic (CFTL)

- *Low-level logic* - easy for software engineers to use.
- *No instrumentation mapping* - formulas have meaning on their own.
- Semantics defined over individual function runs.
- Formulas in CFTL talk about *states* (instantaneous checkpoints) and *transitions* (the computation required to move between states).

Form of CFTL Formulas

- CFTL formulas take prenex normal form

$$\varphi \equiv \forall q_1 \in \Gamma_1, \dots, \forall q_n \in \Gamma_n : \phi(q_1, \dots, q_n)$$

- q_i are variables bound to *states* or *transitions*. Γ_i are *quantification domains*.
- ϕ is a boolean combination of predicates over the q_i and neighbouring states/transitions.

Examples

$$\forall q \in \text{changes}(x) :$$
$$q(x) = \text{True} \implies \text{duration}(\text{next}(q, \text{calls}(f))) < 1$$

$$\forall q \in \text{changes}(y) :$$
$$\forall t \in \text{future}(q, \text{calls}(f)) :$$
$$q(y) = \text{val} \implies \text{duration}(t) \in (0, 0.3)$$

We need to develop

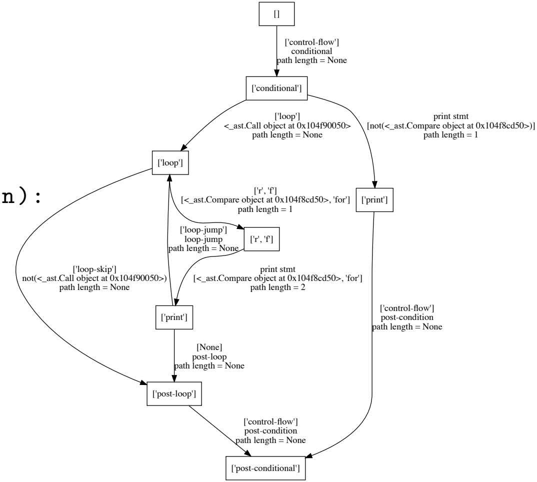
- A *trace* - an abstraction of a run of the program P that we wish to monitor; and
- A *semantics* - a definition of truth of CFTL formulas with respect to our notion of *traces*.

For this, we start by developing a static program model.

Symbolic Control-Flow Graphs (SCFGs)

- For a program P, $\text{SCFG}(P) = \langle V, E, v_s \rangle$.
- V is a set of *symbolic states*. Symbolic states are maps from program variables/functions to $\{\text{undefined}, \text{changed}, \text{unchanged}, \text{called}\}$.
- $E \subset V \times V$ is a set of edges between symbolic states.
- $v_s \in V$ is the starting state.

```
if n > 10:
  for i in range(n):
    r = f(i)
    print(r)
else:
  print("nope")
```



Dynamic Runs as Traces

- *Dynamic Run* $\mathcal{D}$ - finite sequence of *concrete states*

$$\langle t_1, \sigma_1, \tau_1 \rangle, \dots, \langle t_n, \sigma_n, \tau_n \rangle$$

- For timestamps t_i with $t_{i+1} > t_i$, symbolic states σ_i and concrete states τ_i giving concrete values to each $x \in \mathrm{dom}(\sigma_i)$.
- *Transitions* are pairs $\Delta\tau_i = \langle \tau_i, \tau_{i+1} \rangle$.

Properties

- For a concrete state $\langle t, \sigma, \tau \rangle$, time$(\langle t, \sigma, \tau \rangle) = t$.
- For a transition $\Delta\tau = \langle \langle t, \sigma, \tau \rangle, \langle t', \sigma', \tau' \rangle \rangle$, time$(\Delta\tau) = t$.
- The *duration* of $\Delta\tau$ is duration$(\Delta\tau) = t' - t$.

Predicates

- We write predicates over states and transitions from dynamic runs.
- Let $\langle t, \sigma, \tau \rangle$ be a state from a dynamic run $\mathcal{D}$.
- Then we write $\langle t, \sigma, \tau \rangle \vdash \text{changes}(x) \iff \sigma(x) = \text{changed}.$
- Or, for $\Delta\tau = \langle \langle t_i, \sigma_i, \tau_i \rangle, \langle t_{i+1}, \sigma_{i+1}, \tau_{i+1} \rangle \rangle$, $\Delta\tau \vdash \text{calls}(f) \iff \sigma_{i+1}(f) = \text{called}.$

Quantification Domains

- Recall the form of CFTL formulas

$$\varphi \equiv \forall q_1 \in \Gamma_1, \ldots, \forall q_n \in \Gamma_n : \phi(q_1, \ldots, q_n)$$

- A *quantification domain* Γ_i is a set of states and transitions, each satisfying the same predicate.
- Hence, $q \in \Gamma_1$ is abuse of notation for $q \vdash \text{calls}(f)$.

Atoms

- For a CFTL formula φ, let A_φ be the set of atoms. For example:

$$\begin{aligned}\varphi \equiv \forall \boldsymbol{q} \in\ & \text{changes}(x) : \\ & \text{duration}(\text{next}(q, \text{calls}(g))) \in (0, 0.3)\end{aligned}$$

$$A_\varphi = \{\text{duration}(q, \text{calls}(g)) \in (0, 0.3)\}$$

Semantics

$$
\begin{array}{lll}
\mathcal{D}, tr & \vdash & \text{calls}(f) \quad \text{iff} \\
& & \quad \text{for every path } \pi \in \text{paths}(tr) \text{ there is:} \\
& & \quad \text{some } \langle \sigma_1, \sigma_2 \rangle \in \pi \\
& & \quad \text{such that } \sigma_2(f) = \text{called}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{D}, q & \vdash & \text{future}_S(s, \text{changes}(x)) \quad \text{iff} \\
& & \quad \text{time}(q) > \text{time}(s) \text{ and } \mathcal{D}, q \vdash \text{changes}(x)
\end{array}
$$

Semantics

$$
\begin{array}{lcl}
\text{eval}(\mathcal{D}, \theta, q) & = & \theta(q) \\
\text{eval}(\mathcal{D}, \theta, tr) & = & \theta(tr) \\
\text{eval}(\mathcal{D}, \theta, \text{source}(T)) & = & \text{source}(\text{eval}(\mathcal{D}, \theta, T)) \\
\text{eval}(\mathcal{D}, \theta, \text{dest}(T)) & = & \text{dest}(\text{eval}(\mathcal{D}, \theta, T)) \\
\text{eval}(\mathcal{D}, \theta, \text{incident}(S)) & = & \text{incident}(\mathcal{D}, \text{eval}(\mathcal{D}, \theta, S)) \\
\text{eval}\left(\begin{array}{l} \mathcal{D}, \theta, \\ \text{next}_S(X, \text{changes}(x)) \end{array}\right) & = & q \text{ such that:}
\end{array}
$$

$$
\begin{array}{c}
\text{time}(q) > \text{time}(\text{eval}(\mathcal{D}, \theta, X)) \text{ and} \\
\mathcal{D}, q \vdash \text{changes}(x) \text{ and there is no} \\
q' \text{ with } \text{time}(\text{eval}(\mathcal{D}, \theta, X)) < \text{time}(q') < \text{time}(q) \text{ and} \\
\mathcal{D}, q' \vdash \text{changes}(x)
\end{array}
$$

Semantics

$$\begin{array}{lcl}
\mathcal{D}, \theta & \models & \forall^S q \in \Gamma_S : \phi \text{ iff} \\
& & \quad \text{for all } c \in \Gamma_S \text{ we have } \mathcal{D}, \theta[q \mapsto c] \models \phi \\
\mathcal{D}, \theta & \models & \forall^T tr \in \Gamma_T : \phi \text{ iff} \\
& & \quad \text{for all } c \in \Gamma_T \text{ we have } \mathcal{D}, \theta[tr \mapsto c] \models \phi \\
\mathcal{D}, \theta & \models & true \\
\mathcal{D}, \theta & \models & \phi_1 \vee \phi_2 \text{ iff } \mathcal{D}, \theta \models \phi_1 \text{ or } \mathcal{D}, \theta \models \phi_2 \\
\mathcal{D}, \theta & \models & \neg \phi \text{ iff not } \mathcal{D}, \theta \models \phi \\
\mathcal{D}, \theta & \models & S(x) = v \text{ iff } \mathrm{eval}(\mathcal{D}, \theta, S)(x) = v \\
\mathcal{D}, \theta & \models & S(x) \in [n, m] \text{ iff } \mathrm{eval}(\mathcal{D}, \theta, S)(x) \in [n, m] \\
\mathcal{D}, \theta & \models & S(x) \in (n, m) \text{ iff } \mathrm{eval}(\mathcal{D}, \theta, S)(x) \in (n, m) \\
\mathcal{D}, \theta & \models & \mathrm{duration}(T) \in (n, m) \text{ iff} \\
& & \quad \mathrm{duration}(\mathrm{eval}(\mathcal{D}, \theta, T)) \in (n, m) \\
\mathcal{D}, \theta & \models & \mathrm{duration}(T) \in [n, m] \text{ iff} \\
& & \quad \mathrm{duration}(\mathrm{eval}(\mathcal{D}, \theta, T)) \in [n, m]
\end{array}$$

Singly-Quantified Formulas

"Every call to the function f should take less than 5 units of time"

$$\underbrace{\forall t \in \text{calls}(f) :}_{\text{all calls of } f} \text{duration}(t) < 5.$$

With a Dynamic Run

$$\forall t \in \mathrm{calls}(f) : \mathrm{duration}(t) < 5.$$

$$\begin{aligned}\mathcal{D} = \; & \langle 1, [x \mapsto \mathrm{undefined}, f \mapsto \mathrm{undefined}], []\rangle, \\ & \langle 2, [x \mapsto \mathrm{changed}, f \mapsto \mathrm{undefined}], []\rangle, \\ & \langle 8, [x \mapsto \mathrm{unchanged}, f \mapsto \mathrm{called}], []\rangle\end{aligned}$$

FAILURE - the transition $t = \langle 1, [x \mapsto \mathrm{changed}, f \mapsto \mathrm{undefined}], []\rangle, \langle 1, [x \mapsto \mathrm{unchanged}, f \mapsto \mathrm{called}], []\rangle \vdash \mathrm{calls}(f)$ but $\mathrm{duration}(t) = 8 - 2$.

Multiple Quantification

- Using the predicates we have so far, changes(x) and calls(f), singly-quantified formulas are straightforward.
- We use an extra predicate on states or transitions q - future(q, Γ) where Γ is calls or changes.

$$\begin{aligned}
&\forall q \in \mathrm{changes}(x):\\
&\forall t \in \mathrm{future}(q, \mathrm{calls}(f)):\\
&\qquad q(x) = \mathrm{True} \implies \mathrm{duration}(t) < 1
\end{aligned}$$

“Everytime x changes (bound to q), if it’s set to True, then every future call to f (bound to t) should take less than 1 unit of time.”

Multiple Quantification

- Instead of considering *nested quantification*, we consider quantification over a *product space*.

$$\forall \bar{q} \in \Gamma_1 \times \cdots \times \Gamma_n : \phi(\bar{q})$$

- where $\bar{q} = [q_1 \mapsto v_1, \ldots, q_n \mapsto v_n]$ is a *concrete binding* for variables q_i and *states* or *transitions* v_i.
- Each $\bar{q}$ corresponds to an *and-or* formula tree which collapses.

Monitoring

- **The filter problem** - Typical RV approaches imagine the program as a black-box that generates a trace that is not derived from the property being checked.
- **The lookup problem** - Given some data that *is* relevant, how do we decide the bindings/atoms to which it contributes?

The Lookup Problem

- This solution requires that we properly write down an instrumentation algorithm for CFTL.
- To save time, I will only cover the singly-quantified case.

Atom-driven Instrumentation

- General idea: find instructions in the program that *could* generate concrete bindings.
- We do this by recursing over the SCFG to identify vertices or edges which could be a part of the symbolic supports of elements of the quantification domain.
- The resulting set is the *Binding Space*, and denoted by $\mathcal{B}_\varphi$.

Binding Spaces

- A Binding Space $\mathcal{B}_\varphi$ derived from SCFG(P) wrt φ is a set of maps β.
- For each $\beta \in \mathcal{B}_\varphi$, β sends variables from φ to candidates for symbolic supports of states/transitions generated at runtime.
- For example, $\forall q \in \text{changes}(x) : q(x) < 10$ yields a set of maps from q to vertices v with $v(x) = \text{changed}$.

Example

$$\varphi \equiv \forall t \in \text{calls}(f) : \text{duration}(t) < 1$$

$$\mathcal{B}_{\varphi} = \{[t \mapsto \texttt{f(i)}]\}$$

```
for n in range(5):
  f(i)
```

The symbolic support map $s(t)$ on transitions $t \vdash \text{calls}(f)$ cannot be injective.

Symbolic Support wrt Bindings

- For a concrete binding $\bar{q} = [q_1 \mapsto v_1, \ldots, q_n \mapsto v_n]$, the $\beta \in \mathcal{B}_\varphi$ that acts as symbolic support for $\bar{q}$ is the map $[q_1 \mapsto s(v_1), \ldots, q_n \mapsto s(v_n)]$.
- We write $s(\bar{q}) = \beta$.

Atom-driven Instrumentation - singly-quantified

For some CFTL formula $\varphi \equiv \forall q \in \Gamma : \phi(q)$ and some $\mathrm{SCFG}(P) = \langle V, E, v_s \rangle$:

1. Compute $\mathcal{B}_{\varphi}$ recursively using Γ.
2. For each $\beta \in \mathcal{B}_{\varphi}$ with index $i_{\mathcal{B}}$:
    - 2.1 For each $\alpha \in A(\varphi)$ with index i_{α}:
        - 2.1.1 Use α to find neighbouring points around $\beta(q)$ in $SCFG(P)$.

Lookup

- Given $\langle i_{\mathcal{B}}, i_{\alpha} \rangle$ pairs, for $\varphi \equiv \forall q \in \Gamma : \psi(q)$:
- We group formula trees by $i_{\mathcal{B}}$ values.
- Hence, lookup of the monitors (formula trees) to update for each observation is immediate given $i_{\mathcal{B}}$.
- Lookup of the part of the formula tree is also straightforward given i_{α}.

Filtering

- We accidentally solved the filter problem via atom-driven instrumentation!
- Atom-driven instrumentation determines the points in the program that *may* generate observations that we can use to check φ.
- We will never miss an observation, but there are ways in which we can get too much data.
- Current research looks at what we can do to move instrumentation as close to optimality as possible.

VyPR

- This theory was used to build the VyPR tool.
- The initial version ran only on Python programs with respect to single CFTL properties.
- It introduced the PyCFTL library for building CFTL specifications in Python.

PyCFTL

$$\forall q \in \mathsf{changes}(\mathsf{val}) : \\ \qquad \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\mathsf{func}))) \in [0, 3]$$

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

VyPR2 pipeline

1. Engineers describe the performance of their web service in a PyCFTL specification file.
2. Web service is pulled to a production machine.
3. VyPR2 instruments functions according to the PyCFTL specification file.
4. The web service is monitored at runtime.
5. Verdict information is collected on VyPR2's separate server.

Context - LHC and CMS

- The LHC (Large Hadron Collider) is a circular proton-proton collider at CERN in Geneva, Switzerland.
- On the LHC lies the Compact Muon Solenoid (CMS) detector.
- I'm going to describe experience applying VyPR2 on the CMS Experiment.
- It was performed in close collaboration with the Alignment, Calibrations and Databases (AlCaDB) group of the CMS Experiment.

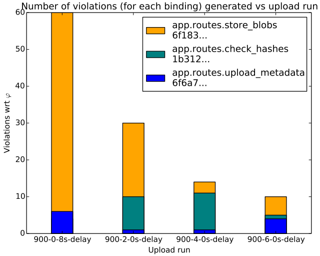
Conditions Upload

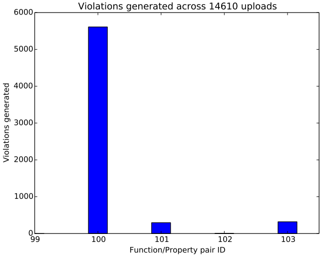
- Before physics analyses can be performed on data taken during LHC runs, *reconstruction* must take place.
- This process requires *Event* and *Non-event* data.
- The Non-event data are so-called Conditions.
- There is a Python-based web service responsible for uploading this to a database after computation.

Simulating LHC Runs

- We cannot safely inject untested verification code into critical infrastructure.
- Instead, with the help of CMS' Alignment and Calibrations group, we recorded Conditions uploads during 6 months.
- The result was a dataset of $\approx 14,600$ Conditions uploads.
- We replayed this dataset in an experimental setup almost identical to the production one.

Results





Unpredictable database latency.

Latency from an optimisation.

$$\forall q \in \text{changes}(\text{hashes}) : \text{duration}(\text{next}(q, \text{calls}(\text{notFound}))) < 0.3$$

Runtime Verification in High Energy Physics

- VyPR is *publicly available* - `http://cern.ch/vypr`.
- While preparing for the High-Luminosity LHC is still a driving force:
- We have a seminar scheduled in *CERN IT*, which will give a chance to find new uses for VyPR across CERN.
- Finally, RV research at CERN addresses the notorious problem of lack of test cases.