Towards Automated Performance Analysis of Python Programs

University of Manchester Formal Methods Seminar

Joshua Heneage Dawes¹,²,³  Giles Reger¹  Giovanni Franzoni²  Andreas Pfeiffer²

¹University of Manchester, Manchester, UK
²CERN, Geneva, Switzerland
³joshua.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.
We wish to check, at runtime, whether some program $P$ holds a property $\varphi$ written in some temporal logic, for example Linear-time Temporal Logic or Metric Temporal Logic.

- A monitor is synthesised for $\varphi$.
- Such a monitor is often an automaton $A_\varphi$.
- Runs of $P$ are abstracted into traces $\tau$, holding enough information to check $\varphi$. 
Practicalities

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

\begin{align*}
p & \iff x < 10 \\
q & \iff \text{call function}
\end{align*}
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

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

- \( q_i \) are variables bound to states or transitions.
- \( \Gamma_i \) are quantification domains.
- \( \phi \) is a boolean combination of predicates over the \( q_i \) and neighbouring states/ transitions.
Examples

∀q ∈ changes(x) :

q(x) = True ⟺ duration(next(q, calls(f))) < 1

∀q ∈ changes(y) :
∀t ∈ future(q, calls(f)) :

q(y) = val ⟺ duration(t) ∈ (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 \{undefined, changed, unchanged, called\}.
- $E \subseteq V \times V$ 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
  \[
  \langle t_1, \sigma_1, \tau_1 \rangle, \ldots, \langle t_n, \sigma_n, \tau_n \rangle
  \]

- For timestamps $t_i$ with $t_{i+1} > t_i$, symbolic states $\sigma_i$ and concrete states $\tau_i$ giving concrete values to each $x \in \text{dom}(\sigma_i)$.

- **Transitions** are pairs $\Delta \tau_i = \langle \tau_i, \tau_{i+1} \rangle$. 

Properties

• For a concrete state \( \langle t, \sigma, \tau \rangle \),
  \( \text{time}(\langle t, \sigma, \tau \rangle) = t \).

• For a transition \( \Delta \tau = \langle \langle t, \sigma, \tau \rangle, \langle t', \sigma', \tau' \rangle \rangle \),
  \( \text{time}(\Delta \tau) = t \).

• The *duration* of \( \Delta \tau \) is \( \text{duration}(\Delta \tau) = t' - t \).
Predicates

• We write predicates over states and transitions from dynamic runs.
• Let $\langle t, \sigma, \tau \rangle$ be a state from a dynamic run $D$.
• 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 \( \Gamma_i \) 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 $\phi$, let $A_{\phi}$ be the set of atoms. For example:

\[
\phi \equiv \forall q \in \text{changes}(x) : \\
\quad \text{duration}(\text{next}(q, \text{calls}(g))) \in (0, 0.3)
\]

\[
A_{\phi} = \{ \text{duration}(q, \text{calls}(g)) \in (0, 0.3) \} 
\]
Semantics

\[ \mathcal{D}, tr \vdash \text{calls}(f) \text{ iff} \]
for every path \( \pi \in \text{paths}(tr) \) there is:

some \( \langle \sigma_1, \sigma_2 \rangle \in \pi \)
such that \( \sigma_2(f) = \text{called} \)

\[ \mathcal{D}, q \vdash \text{future}_S(s, \text{changes}(x)) \text{ iff} \]
time(q) > time(s) and \( \mathcal{D}, q \vdash \text{changes}(x) \)
Semantics

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

\[
\begin{align*}
time(q) &> time(\text{eval}(\mathcal{D}, \theta, X)) \text{ and} \\
\mathcal{D}, q \vdash \text{changes}(x) \text{ and there is no} \\
q' \text{ with } time(\text{eval}(\mathcal{D}, \theta, X)) < time(q') < time(q) \text{ and} \\
\mathcal{D}, q' \vdash \text{changes}(x)
\end{align*}
\]
Semantics

\[ D, \theta \models S \models \Gamma : \phi \text{ iff for all } c \in \Gamma \text{ we have } D, \theta[q \mapsto c] \models \phi \]

\[ D, \theta \models T \models \Gamma : \phi \text{ iff for all } c \in \Gamma \text{ we have } D, \theta[tr \mapsto c] \models \phi \]

\[ D, \theta \models \text{true} \]

\[ D, \theta \models \phi_1 \lor \phi_2 \text{ iff } D, \theta \models \phi_1 \text{ or } D, \theta \models \phi_2 \]

\[ D, \theta \models \neg \phi \text{ iff not } D, \theta \models \phi \]

\[ D, \theta \models S(x) = v \text{ iff } \text{eval}(D, \theta, S)(x) = v \]

\[ D, \theta \models S(x) \in [n, m] \text{ iff } \text{eval}(D, \theta, S)(x) \in [n, m] \]

\[ D, \theta \models S(x) \in (n, m) \text{ iff } \text{eval}(D, \theta, S)(x) \in (n, m) \]

\[ D, \theta \models \text{duration}(T) \in (n, m) \text{ iff } \text{duration}(\text{eval}(D, \theta, T)) \in (n, m) \]

\[ D, \theta \models \text{duration}(T) \in [n, m] \text{ iff } \text{duration}(\text{eval}(D, \theta, T)) \in [n, m] \]
Singly-Quantified Formulas

“Every call to the function $f$ should take less than 5 units of time”

$$\forall t \in \text{calls}(f) : \text{duration}(t) < 5.$$
With a Dynamic Run

∀t ∈ \text{calls}(f) : \text{duration}(t) < 5.

\mathcal{D} = \langle 1, [x \mapsto \text{undefined}, f \mapsto \text{undefined}], [] \rangle,
\langle 2, [x \mapsto \text{changed}, f \mapsto \text{undefined}], [] \rangle,
\langle 8, [x \mapsto \text{unchanged}, f \mapsto \text{called}], [] \rangle

\text{FAILURE} - \text{the transition}
\quad t = \langle 1, [x \mapsto \text{changed}, f \mapsto \text{undefined}], [] \rangle, \langle 1, [x \mapsto \text{unchanged}, f \mapsto \text{called}], [] \rangle \vdash \text{calls}(f) \text{ but } \text{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($q, \Gamma$) where $\Gamma$ is calls or changes.
\[ \forall q \in \text{changes}(x) : \\
\forall t \in \text{future}(q, \text{calls}(f)) : \\
q(x) = \text{True} \implies \text{duration}(t) < 1 \]

“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.

\[ \forall \vec{q} \in \Gamma_1 \times \cdots \times \Gamma_n : \phi(\vec{q}) \]

• where \( \vec{q} = [q_1 \mapsto v_1, \ldots, q_n \mapsto v_n] \) is a concrete binding for variables \( q_i \) and states or transitions \( v_i \).

• Each \( \vec{q} \) 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 $B_\varphi$. 
Binding Spaces

• A Binding Space $B_\varphi$ derived from SCFG$(P)$ wrt $\varphi$ is a set of maps $\beta$.

• For each $\beta \in B_\varphi$, $\beta$ sends variables from $\varphi$ 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

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

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

$$B_\varphi = \left\{ [t \mapsto f(i)] \right\}$$

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 $\beta \in B_\varphi$ 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 $SCFG(P) = \langle V, E, v_s \rangle$:

1. Compute $B_\varphi$ recursively using $\Gamma$.
2. For each $\beta \in B_\varphi$ with index $i_\beta$:
   2.1 For each $\alpha \in A(\varphi)$ with index $i_\alpha$:
      2.1.1 Use $\alpha$ to find neighbouring points around $\beta(q)$ in $SCFG(P)$. 
Lookup

- Given $\langle i_B, i_\alpha \rangle$ pairs, for $\varphi \equiv \forall q \in \Gamma : \psi(q)$:
- We group formula trees by $i_B$ values.
- Hence, lookup of the monitors (formula trees) to update for each observation is immediate given $i_B$.
- Lookup of the part of the formula tree is also straightforward given $i_\alpha$. 
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 $\varphi$.
• 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.
∀q ∈ changes(val) :
duration(next(q, calls(func))) ∈ [0, 3]

Forall(q = changes('val')).\nCheck(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.

∀q ∈ changes(hashes) :
   duration(next(q, calls(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.