VyPR2: A Framework for Runtime Verification of Python Web Services

Joshua Heneage Dawes[1,2](✉), Giles Reger[1], Giovanni Franzoni[2],
Andreas Pfeiffer[2], and Giacomo Govi[3]

[1] University of Manchester, Manchester, UK
[2] CERN, Geneva, Switzerland
joshua.dawes@cern.ch
[3] Fermi National Accelerator Laboratory,
Batavia, IL, USA

Abstract. Runtime Verification (RV) is the process of checking whether a run of a system holds a given property. In order to perform such a check online, the algorithm used to monitor the property must induce minimal overhead. This paper focuses on two areas that have received little attention from the RV community: Python programs and web services. Our first contribution is the VyPR runtime verification tool for single-threaded Python programs. The tool handles specifications in our, previously introduced, Control-Flow Temporal Logic (CFTL), which supports the specification of state and time constraints over runs of functions. VyPR minimally (in terms of reachability) instruments the input program with respect to a CFTL specification and then uses instrumentation information to optimise the monitoring algorithm. Our second contribution is the lifting of VyPR to the web service setting, resulting in the VyPR2 tool. We first describe the necessary modifications to the architecture of VyPR, and then describe our experience applying VyPR2 to a service that is critical to the physics reconstruction pipeline on the CMS Experiment at CERN.

1 Introduction

Runtime Verification [1] is the process of checking whether a run of a system holds a given property (often written in a temporal logic). This can be checked while the system is running (*online*) or after it has run (*post-mortem* or *offline*). Often this is presented abstractly as checking an abstraction of behaviour, captured by a *trace*. This abstract setting often ignores the practicalities of instrumentation and deployment. This paper presents a tool for the runtime verification of Python-based web services that efficiently handles the instrumentation problem and integrates with the widely used web-framework Flask [2]. This work is carried out within the context of verifying web-services used at the CMS Experiment at CERN.

Despite the wealth of existing logics [3–9], in our work [10,11] performing verification of state and time constraints over Python-based web services on the

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part II, LNCS 11428, pp. 98–114, 2019.
https://doi.org/10.1007/978-3-030-17465-1_6

CMS Experiment at CERN we have found that, in most cases, the existing logics operate at a high level of abstraction in relation to the program under scrutiny. This leads to (1) a less straightforward specification process for engineers, who have to think indirectly about their programs; and (2) difficulty writing specifications about behaviour inside functions themselves. These observations led us to develop Control-Flow Temporal Logic [10,11] (CFTL), a logic that has a tight-coupling with the control flow of the program under scrutiny (so operates at a lower level of abstraction which, in our experience, makes writing specifications with it easier for engineers) and is easy to use to specify state and time constraints over single runs of functions.

After the introduction of CFTL (Sect. 2), the first contribution of this paper is a description of the VyPR tool (Sect. 3), which verifies single-threaded Python programs with respect to CFTL specifications. It does this by (1) providing PyCFTL, the Python binding for CFTL, for writing specifications; (2) instrumenting the input program minimally with respect to reachability; and (3) using the resulting instrumentation information to make its online monitoring algorithm more efficient.

Since the development of VyPR as a prototype verification tool for CFTL, we have found that there are, to the best of our knowledge, no frameworks for fully-automated instrumentation and verification of multiple functions in web services with respect to low-level properties. Therefore, the second contribution of this paper is the lifting of CFTL and VyPR to the web service setting in a tool we call VyPR2 (Sect. 4). We present a general infrastructure for the runtime verification of Python-based web services with respect to CFTL specifications. Moving from VyPR to VyPR2 presents a number of challenges, which we discuss in detail. For the moment, we focus on web services that use the Flask framework, a Python framework that allows one to write a web service by writing Python functions to serve as end-points. VyPR2 admits a simple specification process using PyCFTL, performs automatic and optimised instrumentation of the web service under scrutiny, and provides a separate verdict server for collection of verdicts obtained by monitoring CFTL specifications.

Our final contribution is a case study (Sect. 5) applying VyPR2 to the CMS Conditions Upload Service [12], a single-threaded Python-based web service used on the CMS Experiment at CERN. We find that our verification infrastructure induces minimal overhead on Conditions uploads, with experiments showing an overhead of approximately 4.7%. We also find unexpected violations of the specification, one of which has triggered investigations into a mechanism that was designed to be an optimisation but is in danger of adding unnecessary latency. Ultimately, VyPR2 has made analysis of the performance of a critical part of CMS' physics reconstruction pipeline much more straightforward.

2 Control-Flow Temporal Logic (CFTL)

Both of the tools presented in this paper make use of the CFTL specification language [10,11]. We briefly describe this language, focusing on the kinds of

$$\begin{array}{lll}
\phi & := & \forall q \in \Gamma_S : \phi \mid \forall t \in \Gamma_T : \phi \mid \phi \vee \phi \mid \neg\phi \mid \mathit{true} \mid \phi_A \\
\phi_A & := & S(x) = v \mid S(x) = S(x) \mid S(x) \in (n, m) \mid S(x) \in [n, m] \\
 & & \mid \mathsf{duration}(T) \in (n, m) \mid \mathsf{duration}(T) \in [n, m] \\
\Gamma_S & := & \mathsf{changes}(x) \mid \mathsf{future}_S(q, \mathsf{changes}(x)) \mid \mathsf{future}_S(t, \mathsf{changes}(x)) \\
\Gamma_T & := & \mathsf{calls}(f) \mid \mathsf{future}_T(q, \mathsf{calls}(f)) \mid \mathsf{future}_T(t, \mathsf{calls}(f)) \\
S & := & q \mid \mathsf{source}(T) \mid \mathsf{dest}(T) \mid \mathsf{next}_S(S, \mathsf{changes}(x)) \mid \mathsf{next}_S(T, \mathsf{changes}(x)) \\
T & := & t \mid \mathsf{incident}(S) \mid \mathsf{next}_T(S, \mathsf{calls}(f)) \mid \mathsf{next}_T(T, \mathsf{calls}(f))
\end{array}$$

Fig. 1. Syntax of CFTL.

properties it can capture. CFTL is a linear-time temporal logic whose formulas reason over two central types of objects: *states*, instantaneous *checkpoints* in a program's runtime; and *transitions*, the computation that must happen to move between states.

Consider the following property, taken from the case study in Sect. 5:

> *Whenever* `authenticated` *is changed, if it is set to* `True`, *then all future calls to* `execute` *should take no more than 1 second.*

This can be expressed in CFTL as

$$\begin{array}{l}
\forall q \in \mathsf{changes}(\text{authenticated}) : \\
\forall t \in \mathsf{future}(q, \mathsf{calls}(\text{execute})) : \\
\qquad q(\text{authenticated}) = \text{True} \implies \mathsf{duration}(t) \in [0, 1]
\end{array} \tag{1}$$

This first quantifies over the states q in which the program variable `authenticated` is changed and then over the transitions t occurring after that state that correspond to a call of a program function called `execute`. Given this pair of q and t, the specification then states that if `authenticated` is mapped to `True` by q then the duration of the transition t is within the given range.

Syntax. Figure 1 gives the syntax of CFTL. CFTL specifications take prenex form consisting of a list of quantifiers followed by a quantifier-free part. The quantification domains are defined by Γ_S (for states) and Γ_T (for transitions). Terms produced by the S and T cases denote states and transitions respectively. We often drop the S and T subscripts from future and next when the meaning is clear from the context. The quantifier-free part of CFTL formulas is a boolean combination of *atoms* generated by ϕ_A. Let $A(\varphi)$ be the set of atoms of a CFTL formula φ and, for $\alpha \in A(\varphi)$, let $\mathsf{var}(\alpha)$ be the variable on which α is based. In the above example $A(\varphi) = \{q(\text{authenticated}) = \text{True}, \mathsf{duration}(t) \in [0, 1]\}$, $\mathsf{var}(q(\text{authenticated}) = \text{True}) = q$, and $\mathsf{var}(\mathsf{duration}(t) \in [0, 1]) = t$. A CFTL formula is well-formed if it does not contain any free variables (those not captured by a quantifier) and every nested quantifier depends on the previously quantified variable.

```
Forall(q = changes('authenticated')).\
Forall(t = calls('execute', after='q')).\
Check(lambda q, t : (
  If(q('authenticated').equals(True)).then(
    t.duration()._in([0, 1])
  )
))
```

Fig. 2. An example of a CFTL specification written in Python using PyCFTL.

Semantics. The semantics of CFTL is defined over a *dynamic run* of the program. A dynamic run is a sequence of *states* $\tau = \langle \sigma, t \rangle$, where σ is a map (partial functions with finite domain) from program variables/functions to values and $t \in \mathbb{R}^{\geq}$ is a timestamp. Transitions are then pairs $\langle \tau_i, \tau_j \rangle$ for states τ_i and τ_j. The *product quantification domain* over which a CFTL formula is evaluated is derived from the dynamic run using the quantifier list e.g. by extracting all states where some variable changes. Elements of the product quantification domain are maps from specification variables to concrete states/transitions and will be referred to as *concrete bindings*.

3 VyPR

We now present VyPR, which can perform runtime verification on a single Python function with respect to some CFTL specification φ. Further details can be found in a paper [11] and technical report [10], and the tool is available online at http://cern.ch/vypr/.

Tool Workflow. To runtime verify a Python function we follow the following steps. Firstly the property is captured as a CFTL specification using a Python binding called PyCFTL. Given this specification, VyPR instruments the input program so that the monitoring algorithm receives data from any points in the program that could contribute to a verdict. Finally, the modified program will communicate with the monitor at runtime, which will process the observations to produce a verdict.

3.1 Writing CFTL Specifications with PyCFTL

The first step is to write a CFTL specification. Note that such a specification is specific to a particular function being verified as it refers directly to the symbols in that function. For specification we provide PyCFTL, a Python binding for CFTL. Figure 2 shows the PyCFTL specification for the CFTL specification in Eq. 1. A CFTL specification is defined in PyCFTL in two parts:

1. The first part is the quantification sequence. For example, the quantification $\forall q \in \mathsf{changes}(x)$ is given as `Forall(q = changes('x'))`.

2. The second part, the argument to `Check()`, gives the property to be evaluated for each concrete binding in the quantification domain. This is done by specifying a *template* for the specification with a lambda expression (an anonymous function in Python) whose arguments match the variables in the quantification sequence.

3.2 Instrumenting for CFTL

VyPR instruments a Python program for a CFTL specification φ by building up the set Inst containing all points in the program that could contribute to the verdict of φ. VyPR works at the level of the *abstract syntax tree* (AST) of the program and the program points of interest are nodes in the AST. Once this set of nodes has been computed, the AST is modified to add instruments at each of these points.

During runtime monitoring the most expensive operation is usually the lookup of the relevant monitor state that needs to be modified. To make monitoring more efficient, our instrumentation algorithm computes Inst by computing a direct lookup structure that allows the monitoring algorithm to go directly to this state. This structure can be abstractly viewed as a tree, $\mathcal{H}_\varphi$, whose leaves are sets that form a partition of Inst and whose intermediate nodes contain the information required to identify the relevant monitoring state.

The first step in computing $\mathcal{H}_\varphi$ is to construct the *Symbolic Control-Flow Graph* (SCFG) of the body of a (Python) function f.

Definition 1. *A symbolic control-flow graph (SCFG) is a directed graph $\langle V, E, v_s \rangle$ where V is a finite set of symbolic states (maps from all program symbols, e.g. program variables/functions, to a status in $\{$changed, unchanged, called, undefined$\}$), $E \subseteq V \times V$ is a finite set of edges, and $v_s \in V$ is the initial symbolic state.*

The SCFG of a function f is independent of any property φ being checked. Our construction of the SCFG of a program encodes information about state changes (by symbolic states) and reachability (by edges being generated for each state-changing instruction in code), making it an ideal structure from which to derive candidate points for state changes. The SCFG is used to find all symbolic states or edges that *could* generate concrete bindings in the product quantification domain of a formula. For example, if the CFTL specification is $\forall q \in \mathsf{changes}(x) : q(x) < 10$, all symbolic states representing changes to x will be identified as having potential to generate concrete bindings. From this, we construct a set of *static* bindings, which are maps from specification variables to candidate symbolic states/edges in the SCFG. The key distinction between *concrete* and *static* bindings is that static bindings are computed from the SCFG before runtime, and can correspond to zero or more concrete bindings during runtime. We call the set of static bindings the *binding space* for φ with respect to the SCFG and denote it by $\mathcal{B}_\varphi$ with the SCFG implicit. Elements β of $\mathcal{B}_\varphi$ form the top level of the tree $\mathcal{H}_\varphi$.

```
Data: $\varphi$ and the SCFG $\langle V, E, v_s \rangle$ of function $f$
Result: Lookup tree $\mathcal{H}_\varphi$
// Construct $\mathcal{B}_\varphi$
$\mathcal{B}_\varphi = \{\emptyset\}$;
foreach quantified variable $(x_i \in \mathsf{predicate})$ in $\varphi$ in order do
    for $v \in V$ do
        if $v$ is a candidate for predicate then
        $\mathcal{B}_\varphi = \{\beta \cup [x_i \mapsto v] \mid \beta \in \mathcal{B}_\varphi \wedge i > 1 \rightarrow \mathsf{reaches}(\beta(x_{i-1}), v)\}$;
    end
end
// Construct $\mathcal{H}_\varphi$
$\mathcal{H}_\varphi = \emptyset$;
for $\beta \in \mathcal{B}_\varphi$ with index $i_\beta$ do
    for quantified variable $x_i$ in $\varphi$ with index $i_q$ do
        foreach $\alpha \in \{\alpha \in A(\varphi) \mid \mathsf{var}(\alpha) = x_i\}$ with index $i_\alpha$ do
            $\mathcal{H}_\varphi\langle i_\beta, i_q, i_\alpha \rangle \leftarrow \mathsf{lift}(\alpha, \beta(x_i))$;
        end
    end
end
```

Algorithm 1: VyPR's algorithm for construction of the tree $\mathcal{H}_\varphi$.

Once $\mathcal{B}_\varphi$ is constructed, for each $\beta \in \mathcal{B}_\varphi$, VyPR lifts each $\alpha \in A(\varphi)$ (the atoms of φ) from the dynamic context to the SCFG in order to find the relevant symbolic states/edges around the symbolic state/edge $\beta(\mathsf{var}(\alpha))$. This process constructs the second and third levels of the tree $\mathcal{H}_\varphi$: the second level consisting of variables, and the third level of atoms in $A(\varphi)$. The leaves on the fourth level of the tree $\mathcal{H}_\varphi$ are then the subsets of Inst; sets of symbolic states or edges from the SCFG.

Whilst we can abstractly view $\mathcal{H}_\varphi$ as a tree, in practice we represent it as a map from triples $\langle i_\mathcal{B}, i_\forall, i_\alpha \rangle$ to symbolic states/edges of the SCFG where $i_\mathcal{B}$, $i_\forall$ and i_α are indices into the binding space, quantifier list, and set of atoms respectively. An instrument placed in the input program for an atom α, using $\mathcal{H}_\varphi$, contains a triple to identify a subset of Inst and a value obs which is whatever code is required to obtain the value necessary to compute a truth value for α. For example, if the instrument is being placed to record the value of a program variable, obs is the name of the variable which, at runtime, is evaluated to give the value the variable holds. Such an instrument, which pushes its triple and evaluated obs value to a queue to be consumed by the monitoring thread, is placed by modifying the Abstract Syntax Tree (AST) of the program.

Our algorithm for construction of $\mathcal{H}_\varphi$ is Algorithm 1. This makes use of a predicate $\mathsf{reaches}$ which checks whether one symbolic state is reachable from another in the SCFG; and a function $\mathsf{lift}(\alpha, v)$ for $\alpha \in A(\varphi)$ and $v \in V$ which gives the symbolic states reachable from v obtained by lifting α to the static context. With the tree $\mathcal{H}_\varphi$ and binding space $\mathcal{B}_\varphi$ defined, in the next section we present our monitoring approach.

3.3 Monitoring for CFTL

The modified version of the body of f resulting from instrumentation is run alongside VyPR's monitoring algorithm, which consumes data from instruments via a consumption queue populated by the main program thread. Monitoring is performed asynchronously. VyPR's monitoring algorithm involves instantiating a formula tree (an and-or tree) for each binding in the quantification domain of a formula. This algorithm uses the triple $\langle i_{\mathcal{B}}, i_{\forall}, i_{\alpha} \rangle$ and evaluated obs value given by each instrument to perform lookup (to find in which formula trees to update the truth value of a specific atom), decide if new formula trees should be instantiated and compute the truth value of the atom at index i_α in $A(\varphi)$.

Given a CFTL formula $\forall q_1 \in \Gamma_1, \ldots, \forall q_n \in \Gamma_n : \psi(q_1, \ldots, q_n)$, when monitoring one can interpret multiple quantification as single quantification over a product space $\Gamma_1 \times \cdots \times \Gamma_n$. Such a space contains concrete bindings $[q_1 \mapsto v_1, \ldots, q_n \mapsto v_n]$ for states or transitions v_i. Each of these concrete bindings generated at runtime corresponds to a single static binding $\beta \in \mathcal{B}_\varphi$. Using this correspondence, we say that each concrete binding has a *supporting static binding* $\beta \in \mathcal{B}_\varphi$.

Given that monitoring is performed by instantiating a formula tree for each concrete binding in the product quantification domain, the speed of lookup of relevant formula trees is greatly increased by grouping them by the indices of supporting static bindings (determined by $i_{\mathcal{B}}$). Hence, to either update or instantiate formula trees, when information is observed from an instrument that helps to evaluate ψ at some concrete binding, the supporting static binding must be found, giving rise to the requirement for static information during monitoring. During monitoring, lookup of which set of formula trees to use is straightforward since the index $i_{\mathcal{B}}$ is given by the instrument.

Once lookup has been performed, the result is a set of formula trees corresponding to the static binding index $i_{\mathcal{B}}$ received from the instrument. From here, the index i_α is used to determine the atom in $A(\varphi)$ whose truth value (computed using the value given by obs) must be updated in each formula tree.

3.4 Verdict Reports

Once execution has finished, a verdict report is generated, which VyPR keeps in memory. Since each formula tree corresponds to a single concrete binding, verdicts share concrete bindings' correspondence with static bindings. Hence, verdicts can be grouped by the supporting static bindings. Given the binding space $\mathcal{B}_\varphi$ computed during instrumentation, a verdict report $\mathcal{V}$ from a single run of a function can be seen as a partial function

$$\mathcal{V} : \mathcal{B}_\varphi \rightarrow (\{\top, \bot\} \times \mathbb{R}_{\geq})^*,$$

sending a static binding $\beta \in \mathcal{B}_\varphi$ to a sequence of pairs containing a verdict from $\{\top, \bot\}$ and a timestamp (the time at which the verdict was obtained). The map $\mathcal{V}$ sends static bindings to sequences of pairs, rather than single pairs, because single static bindings can support multiple concrete bindings, generating multiple verdicts. This is the case if, for example, the static binding is inside a loop that iterates more than once at runtime.

4 An Architecture for Web Service Verification

We begin our description of the architecture of VyPR2, the extension of VyPR to web services, by isolating a number of requirements imposed by web service deployment environments, and production software environments in general, that must be met.

The environment at CERN inside which our verification infrastructure must function is similar to most production environments. It consists of machines for development and production, with each machine automatically pulling the relevant tags from a central repository once engineers have pushed their (locally-tested) code. Based on this deployment architecture, and the architecture of web services, requirements for our Runtime Verification framework include:

Centralised specifications over multiple functions with multiple properties. It should be possible to verify each function in a web service with respect to multiple properties. Further, specifications for the whole web service should be written in a single file, to minimise intrusion into the web service's code.

Making instrumentation data persistent. Web services' code can be pulled from a repository onto a production server and, once launched, be restarted multiple times between successive deployments of different code versions. Therefore, instrumentation data must be persistent between processes.

Persistent verdict data. Similarly, verdict data must be persistent and, furthermore, engineers must be able to perform offline analysis of the verdicts reached by web services at runtime.

An architecture that meets these requirements is illustrated in Fig. 3, and described in the following sections. The resulting tool, VyPR2, will soon be publicly available from http://cern.ch/vypr.

4.1 Specifying Multiple Function, Multiple Property Specifications

For simplicity of use, we have opted to have engineers write their entire specification in a central configuration file, in the root directory of their web service. This is a file written in Python, specifying CFTL properties over the service using the PyCFTL library.

Part of such a configuration file, using the PyCFTL specification given in Fig. 2, is shown in Fig. 4: one must first give the fully-qualified name of the module in the service in standard Python *dot* notation and then, for each function, the list of properties built up using PyCFTL.

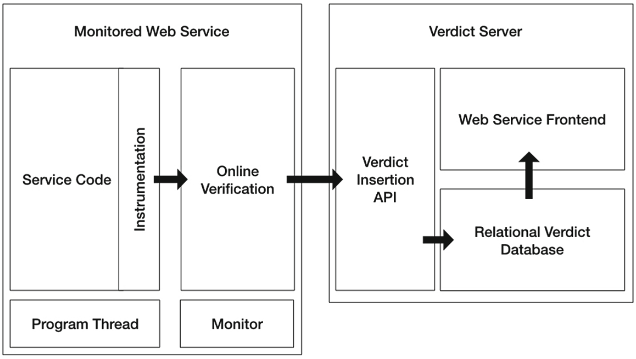

Fig. 3. The architecture of VyPR extended to web services.

$$\varphi_{\mathsf{auth}} \equiv \left(\begin{array}{l} \forall q \in \mathsf{changes}(\text{authenticated}) : \\ \forall t \in \mathsf{future}(q, \mathsf{calls}(\text{execute})) : \\ \qquad q(\text{authenticated}) = \text{True} \implies \mathsf{duration}(t) \in [0, 1] \end{array} \right)$$

```
"app.metadata_handler" : {
  "MetadataHandler.__init__" : [
    Forall(q = changes('authenticated')).\
    Forall(t = calls('execute', after='q')).\
    Check(lambda q, t : (
      If(q('authenticated').equals(True)).then(
        t.duration()._in([0, 1])
      )
    ))
  ]
}
```

Fig. 4. A CFTL specification and its PyCFTL equivalent.

4.2 Instrumentation

Given a specification such as that in Fig. 4, VyPR's strategy must be extended to the multiple function, multiple property context. Multiple functions are dealt with by constructing the SCFG for each function found in the specification and performing instrumentation for each property.

Instrumentation for each property over the same function is performed sequentially: VyPR2 instruments using the AST of the input code, and so instrumentation for each property progressively modifies the AST.

We now describe the modifications required to the actual instruments. In VyPR's simplified setting, instruments need only send the $\langle i_{\mathcal{B}}, i_{\forall}, i_{\alpha} \rangle$ triple along with the obs value relevant to the atom for which the instrument was placed. The multiple function, multiple property setting yields several problems that are solved by modifying existing instruments and adding a new kind.

In our architecture, monitoring is performed by a single thread, which means that this thread must have a way to distinguish between instruments received from different functions. We accomplish this by adding the name of the function to all instruments added to code. By adding the name of the function to all instruments, we deal not only with multiple functions, but with monitored functions calling other monitored functions, in which case monitor states for multiple functions must be maintained at the same time.

We deal with multiple properties over the same function by adding a unique identifier of a property to each of its instruments. We compute a uniquely identifying string for each property by taking the SHA1 hash of the combination of the quantification sequence and the template. We add this unique identifier to each instrument, giving the monitoring algorithm a way to distinguish properties.

Taking the original triple $\langle i_{\mathcal{B}}, i_{\forall}, i_{\alpha} \rangle$, the appropriate obs code, and the new requirements for the function name and the property hash, the new form of instruments that are placed by VyPR2 is $\langle \mathsf{function}, \mathsf{hash}, \mathsf{obs}, i_{\mathcal{B}}, i_{\forall}, i_{\alpha} \rangle$.

4.3 Making Instrumentation Data Persistent

The tree $\mathcal{H}_{\varphi}$ is dependent on the CFTL formula φ for which it has been computed. Hence, if the specification for a given function in the web service consists of a set $\bar{\varphi} = \{\varphi_1, \dots, \varphi_n\}$ of CFTL formulas, the data required to monitor each property at the same time over the same execution of the given function consists of the set of maps $\mathcal{H}_{\varphi_i}$ which can be identified by φ_i. In particular, when data is received from an instrument by the monitoring algorithm, we can assume from Sect. 4.2 that it will contain a unique identifier for the formula for which it was placed. Therefore, the correct tree $\mathcal{H}_{\varphi_i}$ can be determined for each instrument.

We make such instrumentation data persistent by creating new directories in the root of the web service called `binding_spaces` and `instrumentation_maps` to hold the binding spaces and trees, respectively, computed for each function/CFTL property combination. To dump the binding spaces and hierarchy functions in files in these directories, we use Python's `pickle` [13] module.

4.4 Activating Verification in a Web Service

Our infrastructure is designed to minimise intrusion, both by minimising the amount of instrumentation performed and by minimising the amount of code engineers must add to their services for verification to be performed.

With the Flask-based implementation of VyPR2 that we present here, one can *activate* verification by adding the lines `from vypr import Verification` and `verification = Verification(app)` where `app` is the Flask application object required when building a web service with the Flask framework.

Running `verification = Verification(app)` will start up the separate monitoring thread, similar to VyPR, and will also read the serialised binding spaces and trees from the directories described in Sect. 4.3. It will subsequently place them in a map $\mathcal{G}$ from $\langle \text{module.function}, \text{property hash} \rangle$ pairs to objects containing the unserialised forms of the binding spaces and trees.

4.5 A Modified Monitoring Algorithm

VyPR's algorithm uses the tuple $\langle i_{\mathcal{B}}, i_{\forall}, i_{\alpha} \rangle$ with $\mathcal{H}_{\varphi}$ to determine the set of formula trees to update. In this case, $\mathcal{H}_{\varphi}$ is fixed. However, in the web service setting, the additional information regarding the current function that has control and the property to update is present and required to find the correct binding space and tree given by $\mathcal{G}$. From here the process is the same as that used by VyPR, since the monitoring problem has once again collapsed to monitoring a single property over a single function.

4.6 A Verdict Server

For a CFTL formula $\forall q_1 \in \Gamma_1, \ldots, \forall q_n \in \Gamma_n : \psi(q_1, \ldots, q_n)$ over a function f, we use *verdicts* to refer to the sequence of truth values in $(\{\top, \bot\} \times \mathbb{R}^{\geq})^*$, where $\psi(q_1, \ldots, q_n)$ generates a truth value in $\{\top, \bot\}$ for each binding in $\Gamma_1 \times \cdots \times \Gamma_n$ at a time $t \in \mathbb{R}^{\geq}$. To store such verdicts from a specification written over a web service, we now present the most substantial modification to VyPR's architecture: a central server to collect verdicts. This is, in itself, a separate system; communication with it takes place via HTTP. It consists of two major components:

- The server, a Python program that provides an API both for verdict insertion by the monitoring algorithm and for querying by a front-end for verdict visualisation.
- A relational database whose schema is derived from that of the tree $\mathcal{H}_{\varphi}$.

We omit further discussion of the server and first state some facts regarding our relational schema. Functions and properties are paired, so multiple properties over a single function yield multiple pairs; HTTP requests are used to group function calls; function calls correspond to function/property pairs; and verdicts are organised into bindings belonging to a function/property pair. With these facts in mind, one can answer questions such as:

- "For a given HTTP request, function and property φ combination, what were the verdicts generated by monitoring φ across all calls?"
- "For a given verdict and subsystem, which function/property pairs generated the verdict?"
- "For a given function call and verdict, which lines were part of bindings that generated this verdict while monitoring some property φ?"

5 An Application: The CMS Conditions Uploader

We now present the details of the application of VyPR2 to the CMS Conditions Upload Service. We begin by introducing the data with which the CMS Conditions Upload Service works. We then give a brief overview of the existing performance analysis approaches taken at CERN, before describing our approach for replaying real data from LHC runs. Finally, we give our specification and present an analysis of the verdicts derived by monitoring the Conditions Uploader with input taken from our test data, consisting of in the order of 10^4 inputs recorded during LHC runs.

5.1 Conditions Data, Their Computation and Upload

CERN is home to the Large Hadron Collider (LHC) [14], the largest and most powerful particle accelerator ever built. At one of the interaction points on the LHC beamline lies the Compact Muon Solenoid (CMS) [15], a general purpose detector which is a composite of sub-detector systems. Physics analysis at CERN requires reconstruction; a process whose input consists of both Event (collisions) and Non-Event (alignment and calibrations, or Conditions) data. The lifecycle of Conditions data begins with its computation during LHC runs, and ends with its upload to a central Conditions database. The service responsible for this upload is the CMS Conditions Upload service, a precise understanding of the performance of which is vital given planned upgrades to the LHC that will increase the amount of data taken.

The Conditions data used in reconstruction by CMS must define (1) the alignment and calibrations constants associated with a particular subdetector of CMS and (2) the time (run of the LHC) during which those constants are valid. The atomic unit of Conditions is the *Payload*, which is a serialised C++ class whose fields are specific to the subdetector of CMS to which the class corresponds. We define when a Payload applies to the subdetector by associating with it an *Interval of Validity* (IOV). We then group IOVs into sequences by defining *Tags*, which define to which subdetector each Payload associated with the IOVs it contains applies.

The CMS Conditions Uploader is used for release of Conditions by the automated Conditions computation that takes place at Tier 0 [16] (CERN's local computing grid) and detector experts who require their own Conditions. The Uploader is responsible for checking whether the Conditions proposed are valid before inserting the Conditions into the central database.

5.2 A Specification

We now give the specification with which we tested the Upload service on the upload data we collected, along with an interpretation for each property. These were written in collaboration with engineers working on the service.

1. app.usage.Usage.new_upload_session

$$\begin{aligned} &\forall q \in \mathsf{changes}(\text{authenticated}) : \\ &\forall t \in \mathsf{future}(q, \mathsf{calls}(\text{execute})) : \\ &\qquad \begin{pmatrix} q(\text{authenticated}) = \text{True} \\ \implies \mathsf{duration}(t) \in [0, 1] \end{pmatrix} \end{aligned}$$

Whenever `authenticated` *is changed, if it is set to* `True`*, then all future calls to* `execute` *should take no more than 1 second.*

2. app.routes.check_hashes

$$\forall q \in \mathsf{changes}(\text{hashes}) : \mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\text{find\_new\_hashes}))) \in [0, 0.3]$$

When the variable `hashes` *is assigned, the next call to* `find_new_hashes` *should take no more than 0.3 seconds.*

3. app.routes.store_blobs

$$\forall t \in \mathsf{calls}(\text{con.execute}) : \mathsf{duration}(t) \in [0, 2]$$

Every call to the `con.execute` *method on the current database connection should take no more than 2 seconds.*

4. app.metadata_handler.MetadataHandler.__init__

$$\forall t \in \mathsf{calls}(\text{insert\_iovs}) : \mathsf{duration}\left(\begin{array}{l}\mathsf{next}(t,\\ \quad \mathsf{calls}(\text{commit}))\end{array}\right) \in [0, 1]$$

Every time the method `insert_iovs` *is called, the next commit after the insertion should take no more than 1 second.*

5. app.routes.upload_metadata

$$\forall t \in \mathsf{calls}(\text{MetadataHandler}) : \mathsf{duration}(t) \in [0, 1]$$

Every time `MetadataHandler` *is instantiated, the instantiation should take no more than 1 second.*

5.3 Analysis of Verdicts

We present our analysis of the Conditions uploader with respect to the specification in Sect. 5.2. The analysis is performed in two parts:

1. *Complete Replay* - performing a complete upload replay of 14,610 uploads collected over a period of 7 months. The time between uploads in this part is fixed.
2. *Single Tag Replay* - performing a smaller upload replay of $\approx$ 900 uploads based on a single Tag. This part is a subset of the first, but where the time between uploads is varied.

Complete Replay. Figure 5 shows the results of monitoring our specification over a dataset of 14,610 uploads. The x axis is function/property pair IDs from the verdict database snapshot used to generate the plot. The ID to property correspondence is such that ID 99 refers to property 1; ID 100 to property 2; ID 101 to property 3; ID 102 to property 4; and ID 103 to property 5. Clearly, from this plot, the violations of property 2 exceed those caused by other properties by an order of magnitude. The `check_hashes` function carries out an optimisation that we call *hash checking*, used to make sure that a Conditions upload only sends the Payloads that are not already in the target Conditions database. This

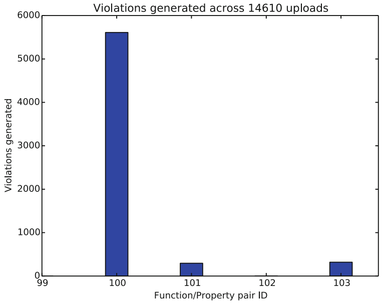

Fig. 5. A plot of number of violations vs properties in the specification, monitored over 14,610 uploads.

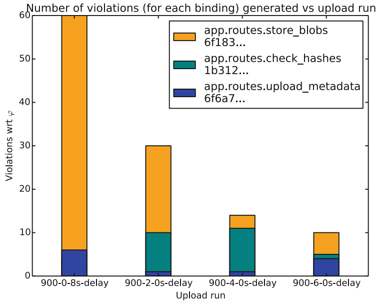

Fig. 6. A plot of violations of parts of our specification vs the replay of the 900 upload dataset. (Color figure online)

is possible because Payloads are uniquely identifiable by their hashes. This optimisation reduces the time spent on Payload uploads by an order of magnitude [12], but the frequency of violation in Fig. 5 suggests that the optimisation itself may be causing unacceptable latency.

Single Tag Replay. Figure 6 shows the results of monitoring a subset of our specification over a dataset of ≈ 900 uploads from a single Tag in the Conditions database. In this case, the x axis is runs of this upload dataset performed with varying delays between uploads, and the y axis is the number of violations based on a specification with 3 properties. This plot is of interest because, for the ≈ 300 Payloads inserted during this replay, it shows that the latency experienced by those insertions (in terms of violations of property 3, shown in orange) decreases as the delay between uploads increases.

5.4 Resulting Investigation

Based on the observations presented in Sect. 5.3, we have made investigation of the number of violations caused by *hash checking* a priority. It is recognised that this process is required, and its addition to the Conditions Uploader was a significant optimisation, but the optimisation can only be considered as such if it does not introduce unacceptable overhead to the upload process.

It is also clear that we should understand the pattern of violations in Fig. 6 more precisely. Given that the Conditions Uploader must operate successfully with both the current and upgraded LHC, it is a priority to understand the behaviour of the Uploader under varying frequencies of uploads. We suspect that investigation into the pattern seen in Fig. 6 will result in modification of either the Conditions Uploader's code, or the way in which Conditions are sent for upload during LHC runs.

5.5 Performance

We now describe the time and space overhead induced by using VyPR2 to monitor the specification in Sect. 5.2 over the Conditions Uploader. We consider both the time overhead on a single upload, and the space required to store intermediate instrumentation data.

To measure the time overhead induced over a single upload, we found that measuring overhead by running our complete upload dataset in a small period of time resulted in erratic database latency (the dataset was recorded over 7 months), so we opted to run a single upload 10 times with and without monitoring. This provided a more realistic upload scenario, and allowed us to see the overhead induced with respect to a single upload process (the process varies depending on the Conditions being uploaded). The result, from 10 runs of the same upload, was an average time overhead of 4.7%. Uploads are performed by a client sending the Conditions to the upload server over multiple HTTP requests, so this overhead is measured starting from when the first request is received by the upload server to when the last response is sent.

The space required to store all of the necessary instrumentation data for the specification in Sect. 5.2 is divided into space for *binding spaces* ($\mathcal{B}_\varphi$), *instrumentation maps* ($\mathcal{H}_\varphi$) and indices (a map from property hashes to the position in the specification at which they are found). The binding spaces took up 170 KB, the instrumentation maps 173 KB and the index map 4.3 KB, giving a total space overhead for instrumentation data storage of 347.3 KB.

6 Related Work

To the best of our knowledge, there is no existing work on Runtime Verification of web services. We are also unaware of other (available and maintained) RV tools for Python (there is Nagini [17], but this focuses on static verification) as most either operate offline (on log files) or focus on other languages such as Java [5,7,18] using AspectJ for instrumentation, C [19], or Erlang [20]. Few RV tools consider the instrumentation problem within the tool. The main exception is Java-MaC [3] who also use the specification to rewrite the Java code directly.

High-Energy Physics. In High Energy Physics, any form of monitoring concentrates on instrumentation in order to carry out manual inspection. For example, the instrumentation and subsequent monitoring of CMS' PhEDEx system for transfer of physics data was performed [21] and resulted in the identification of areas in which latency could be improved. Closer to the case study we present here, CMS uses the pclMon tool to monitor Conditions computation [22]. Finally, the Frontier query caching system performs offline monitoring by analysing logs [23]. None of these approaches uses a formal specification language, and they all collect a single type of statistics for a single defined use case. On the contrary, VyPR2 is *configurable* in the sense that one can change the specification being checked using our formal specification language, CFTL.

7 Conclusion

We have introduced the VyPR tool for monitoring single-threaded Python programs with respect to CFTL specifications, expressed using the PyCFTL library for Python. We then highlighted the problems that one must solve to extend VyPR's architecture to the web service setting, and presented the VyPR2 framework which implements our solutions. VyPR2 is a complete Runtime Verification framework for Flask-based web services written in Python; it provides the PyCFTL library for writing CFTL specifications over an entire web service, automatic minimal (with respect to reachability) instrumentation and efficient monitoring. Finally, we have described our experience using VyPR2 to analyse performance of the CMS Conditions Uploader, a critical part of the physics reconstruction pipeline of the CMS Experiment at CERN.

With the large amount of test data we have at CERN, we plan to extend VyPR2 to address explanation of violations of any part of a specification. This has been agreed within the CMS Experiment as being a significant step in developing the necessary software analysis tools ready for the upgraded LHC.

References

1. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
2. Flask for Python. http://flask.pocoo.org
3. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: a run-time assurance approach for Java programs. Form. Methods Syst. Des. **24**(2), 129–155 (2004)
4. Havelund, K., Reger, G.: Runtime verification logics a language design perspective. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 310–338. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_16
5. Meredith, P.O.N., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT **14**(3), 249–289 (2012)
6. Havelund, K.: Rule-based runtime verification revisited. STTT **17**(2), 143–170 (2015)
7. Colombo, C., Pace, G.J.: Industrial experiences with runtime verification of financial transaction systems: lessons learnt and standing challenges. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 211–232. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_7
8. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_35
9. Basin, D., Krstić, S., Traytel, D.: Almost event-rate independent monitoring of metric dynamic logic. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 85–102. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_6

10. Dawes, J.H., Reger, G.: Specification of State and Time Constraints for Runtime Verification of Functions (2018). arXiv:1806.02621
11. Dawes, J.H., Reger, G.: Specification of temporal properties of functions for runtime verification. In: The 34th ACM/SIGAPP Symposium on Applied Computing (2019)
12. Dawes, J.H., CMS Collaboration: A Python object-oriented framework for the CMS alignment and calibration data. J. Phys.: Conf. Ser. **898**(4), 042059 (2017)
13. `Pickle` for Python. https://docs.python.org/2/library/pickle.html
14. Evans, L., Bryant, P.: LHC machine. J. Instrum. **3**(08), S08001 (2008)
15. The CMS Collaboration: The CMS experiment at the CERN LHC. J. Instrum. **3**(08), S08004 (2008)
16. Britton, D., Lloyd, S.L.: How to deal with petabytes of data: the LHC Grid project. Rep. Progress Phys. **77**(6), 065902 (2014)
17. Eilers, M., Müller, P.: Nagini: a static verifier for python. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 596–603. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_33
18. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 596–610. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_55
19. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language, version 1.5-4, March 2014. frama-c.com/download/e-acsl/e-acsl.pdf
20. Cassar, I., Francalanza, A., Attard, D.P., Aceto, L., Ingólfsdóttir, A.: A suite of monitoring tools for Erlang. In: An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, RV-CuBES 2017, 15 September 2017, Seattle, WA, USA, pp. 41–47 (2017)
21. Bonacorsi, D., Diotalevi, T., Magini, N., Sartirana, A., Taze, M., Wildish, T.: Monitoring data transfer latency in CMS computing operations. J. Phys.: Conf. Ser. **664**(3), 032033 (2015)
22. Oramus, P., et al.: Continuous and fast calibration of the CMS experiment: design of the automated workflows and operational experience. J. Phys.: Conf. Ser. **898**(3), 032041 (2017)
23. Blumenfeld, B., Dykstra, D., Kreuzer, P., Ran, D., Wang, W.: Operational experience with the frontier system in CMS. J. Phys.: Conf. Ser. **396**(5), 052014 (2012)