Specification of Temporal Properties of Functions for Runtime Verification

SAC-SVT 2019 - Limassol - Cyprus

Joshua Heneage Dawes[1,2] Giles Reger[1]

[1]University of Manchester, Manchester, UK

[2]CERN, Geneva, Switzerland

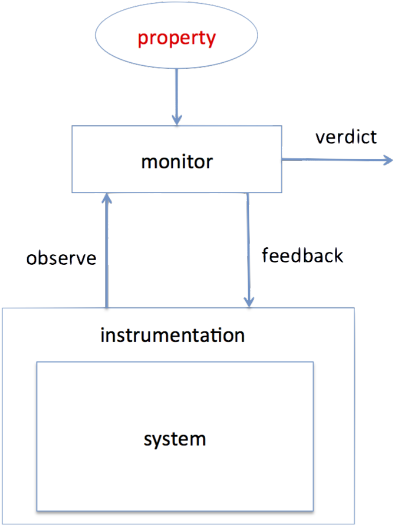
Runtime Verification

Usually: given a run of a system τ and a property we want the system to have ϕ check whether $\tau \in \mathcal{L}(\phi)$ e.g. whether the run is in the language of/satisfies the property.

Pragmatically: instrument the system to produce τ, create a monitor from ϕ to observe τ and decide (at runtime) $\tau \in \mathcal{L}(\phi)$

In this work: given a program P and a property ϕ over constructs in P, instrument P to generate a *sufficiently informative* run τ and then check whether $\tau \in \mathcal{L}(\phi)$

The RV Picture



Motivation 1

```
def process(value,quick):
  if not quick:
    rebalance()
  if newValue(value):
    balanceIns(value)
  result = search(value)
  logging.log(result)
  update(value,result)
  return result
```

$$\square\left(\begin{array}{l} \mathsf{quick} \rightarrow \\ \left(\begin{array}{l} (\neg\mathsf{proc}\ \mathcal{U}_{[0,5]}\ \mathsf{out}) \\ \wedge\ \Diamond_{10}\ \mathsf{fin} \end{array}\right) \end{array}\right)$$

$$\begin{array}{lll} \mathsf{quick} & \leftrightarrow & \textbf{call}\ \mathtt{process} \\ & & \quad \textit{with}\ \mathtt{quick} = 1 \\ \mathsf{proc} & \leftrightarrow & (\textbf{call}\ \mathtt{rebalance}) \vee \\ & & (\textbf{call}\ \mathtt{balanceIns}) \\ \mathsf{out} & \leftrightarrow & \textbf{call}\ \mathtt{logging.log} \\ \mathsf{fin} & \leftrightarrow & \textbf{return}\ \mathtt{process} \end{array}$$

Motivation 2

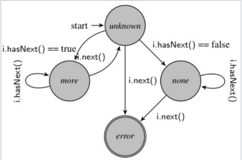
Tool	Specification							Monitor			Deployment					Reaction		Trace				
		explicit																				
				time																		
	implicit	data	output	logical	physical	modality	paradigm	decision procedure	generation	execution	stage	synchronisation	architecture	placement	instrumentation	active	passive	information	sampling	evaluation	precision	model
Aerial	none	p	s	tot	$\mathbb{N}$	all	d	dynamic programming	i	i	on	none	c	ou	none	one	so	e	et	p	p	i
ARTiMon	none	s	s	tot	$\mathbb{N}\mathbb{R}$	all	d	?	i	i	on	none	c	ou	none	one	so	e	et	i	p	i
BeepBeep	none	c	s	tot	none	f	all	stream-processing	i	d	on	all	c	ou	sw	one	so	e	et	p	p	i
DANA	none	p	s	tot	$\mathbb{R}$	all	o	?	i	d	on	sync	c	?	?	one	so	e	et	p	p	f
detectEr	none	s	v	par	none	f	d	dynamic programming	e	i	on	all	c	in	sw	one	so	e	et	p	p	i
E-ACSL	ms	na	r	?	na	na	o	code rewriting with assertions	e	d	on	sync	c	in	sw	e	e	s	na	na	na	na
JavaMOP	none	s	w	tot	none	all	all	trace slicing plugin-based	e	d	on	sync	c	in	swAJ	r	e	e	et	p	p	f
jUnitRV	none	s	v	tot	none	f	d	automata-based (modulo theories eg. SMT solver)	e	d	on	sync	c	in	swR	?	?	e	et	p	p	f
Larva	none	s	v	tot	$\mathbb{N}$	f	o	automata-based	e	d	on	all	c	all	sw	r	so	e	et	p	p	f
LogFire	none	s	w	tot	none	all	o	rewriting-based (RETE)	i	d	all	sync	c	ou	none	one	so	e	et	p	p	f
MarQ/ QEA	none	s	v	tot	$\mathbb{N}$	f	o	automata-based	i	d	all	sync	c	all	sw	one	so	e	et	p	p	f
MonPoly	none	s	s	tot	$\mathbb{N}$	all	d	first-order queries	i	i	on	none	c	ou	none	one	so	e	et	p	p	all
Mufin	none	s	v	tot	none	f	o	automata-based (union-find)	i	d	on	sync	c	ou	none	one	so	e	et	p	p	f
R2U2	none	p	s	tot	$\mathbb{N}$	all	d	automata-based	e	i	on	async	c	ou	none	one	so	e	et	p	p	i
RiTHM	none	p	s	tot	none	f	o	time-triggered runtime verification	e	d	on	async	c	in	sw	one	so	s	all	p	p	i
RTC	ms	na	w	?	na	na	o	?	i	d	on	sync	c	in	sw	r	?	?	et	p	p	na
RV-Monitor	none	s	w	tot	$\mathbb{N}$	all	all	(see JavaMOP)	i	d	all	sync	c	all	sw	r	e	e	et	p	p	f
STePr	none	s	s	tot	$\mathbb{N}$	all	o	?	i	d	on	?	c	ou	none	one	so	e	et	p	p	?
TemPsy/ OCLR-Check	none	p	v	tot	$\mathbb{N}$	all	d	OCL constraint	i	i	off	na	c	ou	none	one	so	e	et	p	p	f
VALOUR	none	s	v	tot	$\mathbb{N}$	all	o	automata-based	i	d	on	all	c	in	swAJ	one	all	e	all	p	p	f

Motivation 2

https://en.wikipedia.org/wiki/Runtime_verification

HasNext [edit]

The Java Iterator interface requires that the `hasNext()` method be called and return true before the `next()` method is called. If this does not occur, it is very possible that a user will iterate "off the end of" a Collection. The figure to the right shows a finite state machine that defines a possible monitor for checking and enforcing this property with runtime verification. From the *unknown* state, it is always an error to call the `next()` method because such an operation could be unsafe. If `hasNext()` is called and returns *true*, it is safe to call `next()`, so the monitor enters the *more* state. If, however, the `hasNext()` method returns *false*, there are no more elements, and the monitor enters the *none* state. In the *more* and *none* states, calling the `hasNext()` method provides no new information. It is safe to call the `next()` method from the *more* state, but it becomes unknown if more elements exist, so the monitor reenters the initial *unknown* state. Finally, calling the `next()` method from the *none* state results in entering the *error* state. What follows is a representation of this property using parametric past time linear temporal logic.



The HasNext Property

$$\forall \text{ Iterator } i \quad i.\mathrm{next}() \rightarrow \odot(i.\mathrm{hasNext}() == \mathit{true})$$

This formula says that any call to the `next()` method must be immediately preceded by a call to `hasNext()` method that returns true. The property here is parametric in the Iterator `i`. Conceptually, this means that there will be one copy of the monitor for each possible Iterator in a test program, although runtime verification systems need not implement their parametric monitors this way. The monitor for this property would be set to trigger a handler when the formula is violated (equivalently when the finite state machine enters the *error* state), which will occur when either `next()` is called without first calling `hasNext()`, or when `hasNext()` is called before `next()`, but returned *false*.

UnsafeEnum [edit]

The Vector class in Java has two means for iterating over its elements. One may use the Iterator interface, as seen in the previous example, or one may use the Enumeration interface. Besides the addition of a remove method for the Iterator interface, the main difference is that Iterator is "fail fast" while Enumeration is not. What this means is that if one modifies the Vector (other than by using the Iterator remove method) when one is iterating over the Vector using an Iterator, a ConcurrentModificationException is thrown. However, when using an Enumeration this is not a case, as mentioned. This can result in non-deterministic results from a program because the Vector is left in an inconsistent state from the perspective of the Enumeration. For legacy programs that still use the Enumeration interface, one may wish to enforce that Enumerations are not used when their underlying Vector is modified. The following parametric regular pattern can be used

```
Vector<String> v = new Vector();
v.add("hello");
v.add("world");
v.add("again");
Enumeration<String> e = v.elements();
String s = "";
v.add("bad!");
while(e.hasMoreElements()){
    s = e.nextElement();
    System.out.println(s);
```

The Separation and Locality Problems

RV approaches often separate instrumentation and specification

- The requirement for an instrumentation mapping can mean that φ cannot be understood straight away, and its exact meaning can even vary depending on the mapping.
- It also means that instrumentation cannot be used to optimise monitoring and the specification cannot be used to optimise instrumentation

RV approaches are often non-local, focussing on interfaces.

- Working with high level properties, rather than properties closely related to P, can be unintuitive for engineers.

This work combines local specification with instrumentation.

This Talk

In this talk I will

- Introduce a useful/necessary program abstraction
- Introduce a new logic (CFTL) that addresses the above issues
- Introduce a simple monitoring algorithm for CFTL
- Show how we (minimally) instrument using the specification
- Describe some experimental results

A Language (subset of Python)

We consider simple programs of the form

$$
\begin{array}{lcl}
\mathit{Program} & := & x = \mathit{expr} \mid \mathit{Program}; \mathit{Program} \mid \\
 & & \text{if } \mathit{expr} \text{ then } \mathit{Program} \text{ (else } \mathit{Program}) \mid \\
 & & \text{while } \mathit{expr} \text{ do } \mathit{Program} \mid \text{for } \mathit{expr} \text{ in } \mathit{Program} \\
\mathit{expr} & := & x \mid f(\mathit{expr}_1, \ldots, \mathit{expr}_n) \mid \mathit{arithExpr} \mid \mathit{boolExpr}
\end{array}
$$

No complex control-flow, no concurrency, an over-approximating view of the heap.

Scope is a single function run (no nested calls, no recursion)

This looks like a subset of many languages, we use Python

Symbolic Control-Flow Graphs

A program point is a node in the AST of a program.

Let Sym be the set of symbols in a program P representing variables and functions.

A symbolic state $\sigma = \langle p, m \rangle$ consists of a program point p and a map m from Sym $\rightarrow \{\text{changed}, \text{called}, \text{undefined}\}$

The Symbolic Control-Flow Graph of a program P is a directed graph $\text{SCFG}(P) = \langle V, E, v_s \rangle$ where

- V is a finite set of symbolic states
- E is a set of edges between V (representing instructions)
- $v_s \in V$ is the starting state

Symbolic Control-Flow Graphs

```
1. a = 10
2. for i in range(n):
3.       f(i)
4. a = 20
5. f(a)
6.
```

$$
\begin{tikzcd}
\arrow[r] & {[]_1} \arrow[r] & {[a \mapsto \text{changed}]_2} \arrow[r] \arrow[d] & {[i \mapsto \text{changed}]_3} \arrow[d, shift right] \\
& & {[]_4} \arrow[d] & {[f \mapsto \text{called}]_2} \arrow[l] \arrow[u, bend right, shift right] \\
& & {[a \mapsto \text{changed}]_5} \arrow[r] & {[f \mapsto \text{called}]_6}
\end{tikzcd}
$$

Constructing SCFG

We define a translation function recursively on the structure of programs. $\mathsf{T}\sigma, P$ gives the set of edges from symbolic state σ given the program P.

For example, we translate an assignment as follows

$$
\begin{aligned}
&\mathsf{T}(\sigma, x = \mathit{expr}; P) = \\
&\quad \{\langle \sigma, \langle p(P), [x \mapsto \text{changed}]\rangle\rangle\} \cup \ \mathsf{T}(\langle p(P), [x \mapsto \text{changed}]\rangle, P) \\
&\text{if } \text{fn}(\mathit{expr}) = \emptyset, \ \text{and} \\
&\quad \{\langle \sigma, \langle p(P), [x_i \mapsto \text{changed}, f_i \mapsto \text{called}]\rangle\rangle\} \\
&\qquad\quad \cup \ \mathsf{T}(\langle p(P), [x_i \mapsto \text{changed}, f_i \mapsto \text{called}]\rangle, P) \\
&\text{for } x_i \in \text{VarR and } f_i \in \ \text{fn}(\mathit{expr}) \text{ otherwise}
\end{aligned}
$$

A Notion of Traces: Dynamic Runs

We define dynamic runs over $\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$

A concrete state $\langle t, \sigma, \tau \rangle$ consists of a timestamp $t \in \mathbb{R}^{\geq}$, a symbolic state $\sigma \in V$, and a valuation τ from Sym to values.

A dynamic run $\mathcal{D}$ is a finite sequence of *concrete states* with strictly increasing timestamps

A transition $\langle \langle t, \sigma, \tau \rangle, \langle t', \sigma', \tau' \rangle \rangle$ is a pair of adjacent concrete states in $\mathcal{D}$, it is *well-formed* if there is path in SCFG between σ and σ', and it is *atomic* if $\langle \sigma, \sigma' \rangle \in E$

A dynamic run is well-formed if every transition is well-formed

A dynamic run is *most-general* if every transition is atomic.

Our Example

```
1.  a = 10
2.  for i in range(n):
3.        f(i)
4.  a = 20
5.  f(a)
6.
```

$$
\begin{tikzcd}
{} \arrow[r] & {[]_1} \arrow[r] & {[a \mapsto \text{changed}]_2} \arrow[r] \arrow[d] & {[i \mapsto \text{changed}]_3} \arrow[d] \\
 & & {[]_4} \arrow[d] & {[f \mapsto \text{called}]_2} \arrow[l] \arrow[u, bend right] \\
 & & {[a \mapsto \text{changed}]_5} \arrow[r] & {[f \mapsto \text{called}]_6}
\end{tikzcd}
$$

Deterministic, so family of dynamic runs differing in timestamps (for a given n)

Our Example

$$
\begin{tikzcd}
{} \arrow[r] & {[]_1} \arrow[r] & {[a \mapsto \text{changed}]_2} \arrow[r] \arrow[d] & {[i \mapsto \text{changed}]_3} \arrow[d] \\
& & {[]_4} \arrow[d] & {[f \mapsto \text{called}]_2} \arrow[l] \arrow[u, bend right] \\
& & {[a \mapsto \text{changed}]_5} \arrow[r] & {[f \mapsto \text{called}]_6}
\end{tikzcd}
$$

$$
\begin{array}{l}
\langle 0, []_1, [] \rangle \\
\langle 0.1, [a \mapsto \text{changed}]_2, [a \mapsto 10] \rangle \\
\langle 0.2, [i \mapsto \text{changed}]_3, [a \mapsto 10, i \mapsto 0] \rangle \\
\langle 0.8, [f \mapsto \text{called}]_2, [a \mapsto 10, i \mapsto 0] \rangle \\
\langle 0.9, [i \mapsto \text{changed}]_3, [a \mapsto 10, i \mapsto 1] \rangle \\
\langle 2.1, [f \mapsto \text{called}]_2, [a \mapsto 10, i \mapsto 1] \rangle \\
\langle 2.2, []_4, [a \mapsto 10] \rangle \\
\langle 2.3, [a \mapsto \text{changed}]_5, [a \mapsto 20] \rangle \\
\langle 3.4, [f \mapsto \text{called}]_6, [a \mapsto 20] \rangle
\end{array}
$$

Control-Flow Temporal Logic

"The calls to function f take less than 5 time units"

$$\forall^T t \in \mathrm{calls}(f) : \mathrm{duration}(t) \in (0, 5).$$

"Whenever x changes, its value remains unchanged until the next call of f", i.e. f always sees every change to x

$$\forall^S q \in \mathrm{changes}(x) : q(x) = \mathrm{source}(\mathrm{next}_T(q, \mathrm{calls}(f)))(x).$$

"Whenever x changes, if its value is in $[0, 5)$, then all future calls to f should take units of time in $(0, 10)$"

$$\forall^S q \in \mathrm{changes}(x) : \forall^T t \in \mathrm{future}_T(q, \mathrm{calls}(f)) : \\ (q(x) \in (0, 5) \lor q(x) \in [0, 1]) \implies \mathrm{duration}(t) \in (0, 10).$$

Control-Flow Temporal Logic

"The calls to function f take less than 5 time units"

$$\forall^T t \in \mathrm{calls}(f) : \mathrm{duration}(t) \in (0,5).$$

"Whenever x changes, its value remains unchanged until the next call of f", i.e. f always sees every change to x

$$\forall^S q \in \mathrm{changes}(x) : q(x) = \mathrm{source}(\mathrm{next}_T(q, \mathrm{calls}(f)))(x).$$

"Whenever x changes, if its value is in $[0,5)$, then all future calls to f should take units of time in $(0,10)$"

$$\forall^S q \in \mathrm{changes}(x) : \forall^T t \in \ \mathrm{future}_T(q, \mathrm{calls}(f)) : \\ (q(x) \in (0,5) \vee q(x) \in [0,1]) \implies \mathrm{duration}(t) \in (0,10).$$

Control-Flow Temporal Logic

"The calls to function f take less than 5 time units"

$$\forall^T t \in \mathrm{calls}(f) : \mathrm{duration}(t) \in (0, 5).$$

"Whenever x changes, its value remains unchanged until the next call of f", i.e. f always sees every change to x

$$\forall^S q \in \mathrm{changes}(x) : q(x) = \mathrm{source}(\mathrm{next}_T(q, \mathrm{calls}(f)))(x).$$

"Whenever x changes, if its value is in $[0, 5)$, then all future calls to f should take units of time in $(0, 10)$"

$$\forall^S q \in \mathrm{changes}(x) : \forall^T t \in \ \mathrm{future}_T(q, \mathrm{calls}(f)) : \\ (q(x) \in (0, 5) \vee q(x) \in [0, 1]) \implies \mathrm{duration}(t) \in (0, 10).$$

Control-Flow Temporal Logic

"The calls to function f take less than 5 time units"

$$\forall^T t \in \mathrm{calls}(f) : \mathrm{duration}(t) \in (0, 5).$$

"Whenever x changes, its value remains unchanged until the next call of f", i.e. f always sees every change to x

$$\forall^S q \in \mathrm{changes}(x) : q(x) = \mathrm{source}(\mathrm{next}_T(q, \mathrm{calls}(f)))(x).$$

"Whenever x changes, if its value is in $[0, 5)$, then all future calls to f should take units of time in $(0, 10)$"

$$\forall^S q \in \mathrm{changes}(x) : \forall^T t \in \mathrm{future}_T(q, \mathrm{calls}(f)) : \\ (q(x) \in (0, 5) \vee q(x) \in [0, 1]) \implies \mathrm{duration}(t) \in (0, 10).$$

Our Example

```
1. a = 10
2. for i in range(n):
3.       f(i)
4. a = 20
5. f(a)
6.
```

$$
\begin{tikzcd}
{} \arrow[r] & {[]_1} \arrow[r] & {[a \mapsto \text{changed}]_2} \arrow[r] \arrow[d] & {[i \mapsto \text{changed}]_3} \arrow[d] \\
& & {[]_4} \arrow[d] & {[f \mapsto \text{called}]_2} \arrow[l] \arrow[u, bend right] \\
& & {[a \mapsto \text{changed}]_5} \arrow[r] & {[f \mapsto \text{called}]_6}
\end{tikzcd}
$$

$$
\forall^S q \in \text{changes}(a) : \\
q(a) \in [0, 20] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]
$$

Syntax

$$
\begin{aligned}
\phi &:= \forall^S q \in \Gamma_S : \phi \mid \forall^T t \in \Gamma_T : \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi_S \mid \phi_T \mid \mathit{true} \\
\phi_S &:= S(x) = v \mid S(x) = S(x) \mid S(x) \in (n, m) \mid S(x) \in [n, m] \\
\phi_T &:= \text{duration}(T) \in (n, m) \mid \text{duration}(T) \in [n, m]
\end{aligned}
$$

$$
\begin{aligned}
\Gamma_S &:= \text{changes}(x) \mid \text{future}_S(q, \text{changes}(x)) \mid \text{future}_S(t, \text{changes}(x)) \\
\Gamma_T &:= \text{calls}(f) \mid \text{future}_T(q, \text{calls}(f)) \mid \text{future}_T(t, \text{calls}(f))
\end{aligned}
$$

$$
\begin{aligned}
S &:= q \mid \text{source}(T) \mid \text{dest}(T) \mid \text{next}_S(S, \text{changes}(x)) \mid \\
&\quad\ \text{next}_S(T, \text{changes}(x)) \\
T &:= t \mid \text{incident}(S) \mid \text{next}_T(S, \text{calls}(f)) \mid \text{next}_T(T, \text{calls}(f))
\end{aligned}
$$

Well-formed if well-sorted, in prenex form, and all variables bound exactly once.

Syntax

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

Well-formed if well-sorted, in prenex form, and all variables bound exactly once.

Syntax

$$
\begin{array}{lcl}
\phi & := & \forall^S q \in \Gamma_S : \phi \mid \forall^T t \in \Gamma_T : \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi_S \mid \phi_T \mid \mathit{true} \\
\phi_S & := & S(x) = v \mid S(x) = S(x) \mid S(x) \in (n, m) \mid S(x) \in [n, m] \\
\phi_T & := & \mathrm{duration}(T) \in (n, m) \mid \mathrm{duration}(T) \in [n, m]
\end{array}
$$

$$
\begin{array}{lcl}
\Gamma_S & := & \mathrm{changes}(x) \mid \mathrm{future}_S(q, \mathrm{changes}(x)) \mid \mathrm{future}_S(t, \mathrm{changes}(x)) \\
\Gamma_T & := & \mathrm{calls}(f) \mid \mathrm{future}_T(q, \mathrm{calls}(f)) \mid \mathrm{future}_T(t, \mathrm{calls}(f))
\end{array}
$$

$$
\begin{array}{lcl}
S & := & q \mid \mathrm{source}(T) \mid \mathrm{dest}(T) \mid \mathrm{next}_S(S, \mathrm{changes}(x)) \mid \\
 & & \mathrm{next}_S(T, \mathrm{changes}(x)) \\
T & := & t \mid \mathrm{incident}(S) \mid \mathrm{next}_T(S, \mathrm{calls}(f)) \mid \mathrm{next}_T(T, \mathrm{calls}(f))
\end{array}
$$

Well-formed if well-sorted, in prenex form, and all variables bound exactly once.

Idea behind Semantics

Formulas define points of interest in the program, which are related to symbolic states in the SCFG, which then relate to some concrete states in a dynamic run.

We quantify over these to produce a set of bindings

The quantifier-free formula is then evaluated for each binding where we use the points of interest to interpret temporal formulas

Points of Interest

A state q (or transition tr) in a dynamic run $\mathcal{D}$ satisfies point of interest Γ if $\mathcal{D}, q \vdash \Gamma$ (or $\mathcal{D}, tr \vdash \Gamma$)

$$
\begin{array}{lll}
\mathcal{D}, \langle t, \sigma, \tau \rangle & \vdash & \text{changes}(x) \quad \text{iff} \quad \sigma(x) = \text{changed} \\
\mathcal{D}, q & \vdash & \text{future}_S(s, \text{changes}(x)) \quad \text{iff} \\
 & & \quad \text{t}(q) > \text{t}(s) \text{ and } \mathcal{D}, q \vdash \text{changes}(x) \\
\mathcal{D}, tr & \vdash & \text{calls}(f) \quad \text{iff} \\
 & & \quad \text{for every path } \pi \in \text{paths}(tr) \text{ there is:} \\
 & & \quad \text{some } \langle \sigma_1, \sigma_2 \rangle \in \pi \\
 & & \quad \text{such that } \sigma_2(f) = \text{called} \\
\mathcal{D}, tr & \vdash & \text{future}_T(s, \text{calls}(f)) \quad \text{iff} \\
 & & \quad \text{t}(tr) > \text{t}(s) \text{ and } \mathcal{D}, tr \vdash \text{calls}(f)
\end{array}
$$

Note $\mathcal{D}$ may not be most general e.g. transitions in $\mathcal{D}$ may relate to sets of paths in SCFG

Points of Interest

A state q (or transition tr) in a dynamic run $\mathcal{D}$ satisfies point of interest Γ if $\mathcal{D}, q \vdash \Gamma$ (or $\mathcal{D}, tr \vdash \Gamma$)

$$
\begin{array}{lcl}
\mathcal{D}, \langle t, \sigma, \tau \rangle & \vdash & \mathsf{changes}(x) \quad \text{iff} \quad \sigma(x) = \mathsf{changed} \\
\mathcal{D}, q & \vdash & \mathsf{future}_S(s, \mathsf{changes}(x)) \quad \text{iff} \\
 & & \quad \mathsf{t}(q) > \mathsf{t}(s) \text{ and } \mathcal{D}, q \vdash \mathsf{changes}(x) \\
\mathcal{D}, tr & \vdash & \mathsf{calls}(f) \quad \text{iff} \\
 & & \quad \text{for every path } \pi \in \mathsf{paths}(tr) \text{ there is:} \\
 & & \quad \text{some } \langle \sigma_1, \sigma_2 \rangle \in \pi \\
 & & \quad \text{such that } \sigma_2(f) = \mathsf{called} \\
\mathcal{D}, tr & \vdash & \mathsf{future}_T(s, \mathsf{calls}(f)) \quad \text{iff} \\
 & & \quad \mathsf{t}(tr) > \mathsf{t}(s) \text{ and } \mathcal{D}, tr \vdash \mathsf{calls}(f)
\end{array}
$$

Note $\mathcal{D}$ may not be most general e.g. transitions in $\mathcal{D}$ may relate to sets of paths in SCFG

Quantification Domains

The quantification domain of a quantified state or transition is simply the states or transitions that satisfy the point of interest.

In $\forall^S q \in \Gamma_S$ the variable q ranges over the states c such that $c \vdash \Gamma_S$. Similarly in $\forall^T \in \Gamma_T$.

We overload Γ_S (and Γ_T) to also stand for this set.

This could be computed by iterating over $\mathcal{D}$ and checking $\vdash \Gamma$ for each state (or transition).

Semantics

$$\begin{array}{lcl}
\mathcal{D}, \beta & \models & \forall^S q \in \Gamma_S : \phi \text{ iff for all } c \in \Gamma_S \text{ we have } \mathcal{D}, \beta[q \mapsto c] \models \phi \\
\mathcal{D}, \beta & \models & \forall^T tr \in \Gamma_T : \phi \text{ iff for all } c \in \Gamma_T \text{ we have } \mathcal{D}, \beta[tr \mapsto c] \models \phi \\
\mathcal{D}, \beta & \models & true \\
\mathcal{D}, \beta & \models & \phi_1 \vee \phi_2 \text{ iff } \mathcal{D}, \beta \models \phi_1 \text{ or } \mathcal{D}, \beta \models \phi_2 \\
\mathcal{D}, \beta & \models & \neg\phi \text{ iff not } \mathcal{D}, \beta \models \phi \\
\mathcal{D}, \beta & \models & S(x) = v \text{ iff } \text{eval}(\mathcal{D}, \beta, S)(x) = v \\
\mathcal{D}, \beta & \models & S_1(x_1) = S_2(x_2) \text{ iff } \text{eval}(\mathcal{D}, \beta, S_1)(x_1) = \text{eval}(\mathcal{D}, \beta, S_2)(x_2) \\
\mathcal{D}, \beta & \models & S(x) \in [n, m] \text{ iff } \text{eval}(\mathcal{D}, \beta, S)(x) \in [n, m] \\
\mathcal{D}, \beta & \models & S(x) \in (n, m) \text{ iff } \text{eval}(\mathcal{D}, \beta, S)(x) \in (n, m) \\
\mathcal{D}, \beta & \models & \text{duration}(T) \in (n, m) \text{ iff } \text{duration}(\text{eval}(\mathcal{D}, \beta, T)) \in (n, m) \\
\mathcal{D}, \beta & \models & \text{duration}(T) \in [n, m] \text{ iff } \text{duration}(\text{eval}(\mathcal{D}, \beta, T)) \in [n, m]
\end{array}$$

Where we evaluate quantifier-free formulas on the dynamic run with respect to a given binding.

Semantics

$$
\begin{array}{lcl}
\mathcal{D}, \beta & \models & \forall^S q \in \Gamma_S : \phi \text{ iff for all } c \in \Gamma_S \text{ we have } \mathcal{D}, \beta[q \mapsto c] \models \phi \\
\mathcal{D}, \beta & \models & \forall^T tr \in \Gamma_T : \phi \text{ iff for all } c \in \Gamma_T \text{ we have } \mathcal{D}, \beta[tr \mapsto c] \models \phi \\
\mathcal{D}, \beta & \models & true \\
\mathcal{D}, \beta & \models & \phi_1 \vee \phi_2 \text{ iff } \mathcal{D}, \beta \models \phi_1 \text{ or } \mathcal{D}, \beta \models \phi_2 \\
\mathcal{D}, \beta & \models & \neg\phi \text{ iff not } \mathcal{D}, \beta \models \phi \\
\mathcal{D}, \beta & \models & S(x) = v \text{ iff } \mathrm{eval}(\mathcal{D}, \beta, S)(x) = v \\
\mathcal{D}, \beta & \models & S_1(x_1) = S_2(x_2) \text{ iff } \mathrm{eval}(\mathcal{D}, \beta, S_1)(x_1) = \mathrm{eval}(\mathcal{D}, \beta, S_2)(x_2) \\
\mathcal{D}, \beta & \models & S(x) \in [n, m] \text{ iff } \mathrm{eval}(\mathcal{D}, \beta, S)(x) \in [n, m] \\
\mathcal{D}, \beta & \models & S(x) \in (n, m) \text{ iff } \mathrm{eval}(\mathcal{D}, \beta, S)(x) \in (n, m) \\
\mathcal{D}, \beta & \models & \mathrm{duration}(T) \in (n, m) \text{ iff } \mathrm{duration}(\mathrm{eval}(\mathcal{D}, \beta, T)) \in (n, m) \\
\mathcal{D}, \beta & \models & \mathrm{duration}(T) \in [n, m] \text{ iff } \mathrm{duration}(\mathrm{eval}(\mathcal{D}, \beta, T)) \in [n, m]
\end{array}
$$

Where we evaluate quantifier-free formulas on the dynamic run with respect to a given binding.

Evaluating Non-Temporal Formulas

Evaluating non-temporal formulas relatively straightforward given some functions operating on states and transitions e.g. $\mathrm{source}(\langle q_1, q_2 \rangle) = q_1$.

$$
\begin{array}{lcl}
\mathrm{eval}(\mathcal{D}, \beta, q) & = & \beta(q) \\
\mathrm{eval}(\mathcal{D}, \beta, tr) & = & \beta(tr) \\
\mathrm{eval}(\mathcal{D}, \beta, \mathrm{source}(T)) & = & \mathrm{source}(\mathrm{eval}(\mathcal{D}, \beta, T)) \\
\mathrm{eval}(\mathcal{D}, \beta, \mathrm{dest}(T)) & = & \mathrm{dest}(\mathrm{eval}(\mathcal{D}, \beta, T)) \\
\mathrm{eval}(\mathcal{D}, \beta, \mathrm{incident}(S)) & = & \mathrm{incident}(\mathcal{D}, \mathrm{eval}(\mathcal{D}, \beta, S))
\end{array}
$$

Evaluating Temporal Formulas

For temporal formulas it is necessary to identify the future point of interest.

$$\text{eval}\left(\begin{array}{l}\mathcal{D}, \beta, \\ \text{next}_S(X, \text{changes}(x))\end{array}\right) = q \text{ such that:}$$

$\text{t}(q) > \text{t}(\text{eval}(\mathcal{D}, \beta, X))$ and $\mathcal{D}, q \vdash \text{changes}(x)$ and there is no q' with $\text{t}(\text{eval}(\mathcal{D}, \beta, X)) < \text{t}(q') < \text{t}(q)$ and $\mathcal{D}, q' \vdash \text{changes}(x)$

$$\text{eval}\left(\begin{array}{l}\mathcal{D}, \beta, \\ \text{next}_T(X, \text{calls}(f))\end{array}\right) = tr \text{ such that:}$$

$\text{t}(tr) > \text{t}(\text{eval}(\mathcal{D}, \beta, X))$ and $\mathcal{D}, tr \vdash \text{calls}(f)$ and there is no tr' with $\text{t}(\text{eval}(\mathcal{D}, \beta, X)) < \text{t}(tr') < \text{t}(tr)$ and $\mathcal{D}, tr' \vdash \text{calls}(f)$

Satisfaction/Violation

Finally, a dynamic run $\mathcal{D}$ *satisfies* a (well-formed, well-defined) CFTL formula ϕ if $\mathcal{D}, [] \models \phi$, otherwise $\mathcal{D}$ *violates* ϕ.

Our Example

$$\begin{tikzcd}
{} \arrow[r] & {[]_1} \arrow[r] & {[a \mapsto \text{changed}]_2} \arrow[r] \arrow[d] & {[i \mapsto \text{changed}]_3} \arrow[d] \\
 & & {[]_4} \arrow[d] & {[f \mapsto \text{called}]_2} \arrow[l] \arrow[u, bend right] \\
 & & {[a \mapsto \text{changed}]_5} \arrow[r] & {[f \mapsto \text{called}]_6}
\end{tikzcd}$$

```
a = 10
for i in range(n):
      f(i)
a = 20
f(a)
```

$$\forall^S q \in \text{changes}(a):$$
$$q(a) \in [0, 20] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]$$

Our Example

$$
\begin{array}{ll}
\langle 0, []_1, [] \rangle & \\
\langle 0.1, [a \mapsto \text{changed}]_2, [a \mapsto 10] \rangle & \in \text{changes}(a) \\
\langle 0.2, [i \mapsto \text{changed}]_3, [a \mapsto 10, i \mapsto 0] \rangle & \\
\langle 0.8, [f \mapsto \text{called}]_2, [a \mapsto 10, i \mapsto 0] \rangle & \in \text{calls}(f) \\
\langle 0.9, [i \mapsto \text{changed}]_3, [a \mapsto 10, i \mapsto 1] \rangle & \\
\langle 2.1, [f \mapsto \text{called}]_2, [a \mapsto 10, i \mapsto 1] \rangle & \in \text{calls}(f) \\
\langle 2.2, []_4, [a \mapsto 10] \rangle & \\
\langle 2.3, [a \mapsto \text{changed}]_5, [a \mapsto 20] \rangle & \in \text{changes}(a) \\
\langle 3.4, [f \mapsto \text{called}]_6, [a \mapsto 20] \rangle & \in \text{calls}(f)
\end{array}
$$

$$
\begin{aligned}
&\forall^S q \in \text{changes}(a) : \\
&\qquad q(a) \in [0, 20] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]
\end{aligned}
$$

Our Example

$$
\begin{array}{ll}
\langle 0, []_1, [] \rangle & \\
\langle 0.1, [a \mapsto \text{changed}]_2, [a \mapsto 10] \rangle & \in \text{changes}(a) \\
\langle 0.2, [i \mapsto \text{changed}]_3, [a \mapsto 10, i \mapsto 0] \rangle & \\
\langle 0.8, [f \mapsto \text{called}]_2, [a \mapsto 10, i \mapsto 0] \rangle & \in \text{calls}(f) \\
\langle 0.9, [i \mapsto \text{changed}]_3, [a \mapsto 10, i \mapsto 1] \rangle & \\
\langle 2.1, [f \mapsto \text{called}]_2, [a \mapsto 10, i \mapsto 1] \rangle & \in \text{calls}(f) \\
\langle 2.2, []_4, [a \mapsto 10] \rangle & \\
\langle 2.3, [a \mapsto \text{changed}]_5, [a \mapsto 20] \rangle & \in \text{changes}(a) \\
\langle 3.4, [f \mapsto \text{called}]_6, [a \mapsto 20] \rangle & \in \text{calls}(f)
\end{array}
$$

$$
\forall^S q \in \text{changes}(a) :
$$

$$
q(a) \in [0, 20] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]
$$

Our Example

$$\begin{array}{ll}
\langle 0, []_1, [] \rangle & \\
\langle 0.1, [a \mapsto \text{changed}]_2, [a \mapsto 10] \rangle & \in \text{changes}(a) \\
\langle 0.2, [i \mapsto \text{changed}]_3, [a \mapsto 10, i \mapsto 0] \rangle & \\
\langle 0.8, [f \mapsto \text{called}]_2, [a \mapsto 10, i \mapsto 0] \rangle & \in \text{calls}(f) \\
\langle 0.9, [i \mapsto \text{changed}]_3, [a \mapsto 10, i \mapsto 1] \rangle & \\
\langle 2.1, [f \mapsto \text{called}]_2, [a \mapsto 10, i \mapsto 1] \rangle & \in \text{calls}(f) \\
\langle 2.2, []_4, [a \mapsto 10] \rangle & \\
\langle 2.3, [a \mapsto \text{changed}]_5, [a \mapsto 20] \rangle & \in \text{changes}(a) \\
\langle 3.4, [f \mapsto \text{called}]_6, [a \mapsto 20] \rangle & \in \text{calls}(f)
\end{array}$$

$$\forall^S q \in \text{changes}(a) :$$
$$q(a) \in [0, 20] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]$$

Our Example

$$
\begin{array}{ll}
\langle 0, []_1, [] \rangle & \\
\langle 0.1, [a \mapsto \text{changed}]_2, [a \mapsto 10] \rangle & \in \text{changes}(a) \\
\langle 0.2, [i \mapsto \text{changed}]_3, [a \mapsto 10, i \mapsto 0] \rangle & \\
\langle 0.8, [f \mapsto \text{called}]_2, [a \mapsto 10, i \mapsto 0] \rangle & \in \text{calls}(f) \\
\langle 0.9, [i \mapsto \text{changed}]_3, [a \mapsto 10, i \mapsto 1] \rangle & \\
\langle 2.1, [f \mapsto \text{called}]_2, [a \mapsto 10, i \mapsto 1] \rangle & \in \text{calls}(f) \\
\langle 2.2, []_4, [a \mapsto 10] \rangle & \\
\langle 2.3, [a \mapsto \text{changed}]_5, [a \mapsto 20] \rangle & \in \text{changes}(a) \\
\langle 3.4, [f \mapsto \text{called}]_6, [a \mapsto 20] \rangle & \in \text{calls}(f)
\end{array}
$$

$\forall^S q \in \text{changes}(a) :$

$$q(a) \in [0, 20] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]$$

Our Example

$$
\begin{array}{ll}
\langle 0, []_1, [] \rangle & \\
\langle 0.1, [a \mapsto \text{changed}]_2, [a \mapsto 10] \rangle & \in \text{changes}(a) \\
\langle 0.2, [i \mapsto \text{changed}]_3, [a \mapsto 10, i \mapsto 0] \rangle & \\
\langle 0.8, [f \mapsto \text{called}]_2, [a \mapsto 10, i \mapsto 0] \rangle & \in \text{calls}(f) \\
\langle 0.9, [i \mapsto \text{changed}]_3, [a \mapsto 10, i \mapsto 1] \rangle & \\
\langle 2.1, [f \mapsto \text{called}]_2, [a \mapsto 10, i \mapsto 1] \rangle & \in \text{calls}(f) \\
\langle 2.2, []_4, [a \mapsto 10] \rangle & \\
\langle 2.3, [a \mapsto \text{changed}]_5, [a \mapsto 20] \rangle & \in \text{changes}(a) \\
\langle 3.4, [f \mapsto \text{called}]_6, [a \mapsto 20] \rangle & \in \text{calls}(f)
\end{array}
$$

$$
\forall^S q \in \text{changes}(a) : \quad q(a) \in [0, 20] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]
$$

A Naive Monitoring Algorithm

Maintain a map M from bindings to formula trees

For each concrete state q_i in $\mathcal{D}$

Update bindings:

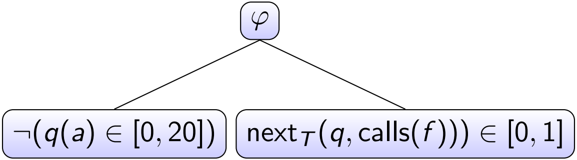
1. If q_i or (q_{i-1}, q_i) are in Γ_1 then create a new binding
2. If there is a binding β that can be extended by q for a quantification domain Γ_i then extend it

Update the formula trees for each binding using q (most will not be updated)

Formula Trees

Simply And-Or trees holding the quantifier-free formulas that can be updated with concrete states to evaluate sub-formulas.

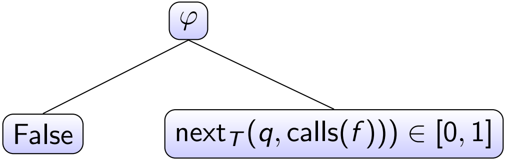
$$\forall^S q \in \mathrm{changes}(a) :$$
$$q(a) \in [0, 20] \implies \mathrm{duration}(\mathrm{next}_T(q, \mathrm{calls}(f))) \in [0, 1]$$



Formula Trees

Simply And-Or trees holding the quantifier-free formulas that can be updated with concrete states to evaluate sub-formulas.

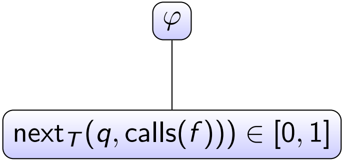
$$\forall^S q \in \text{changes}(a) :$$
$$q(a) \in [0, 20] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]$$



Formula Trees

Simply And-Or trees holding the quantifier-free formulas that can be updated with concrete states to evaluate sub-formulas.

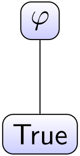
$$\forall^S q \in \text{changes}(a) :$$
$$q(a) \in [0, 20] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]$$



Formula Trees

Simply And-Or trees holding the quantifier-free formulas that can be updated with concrete states to evaluate sub-formulas.

$$\forall^S q \in \mathsf{changes}(a) :$$
$$q(a) \in [0, 20] \implies \mathsf{duration}(\mathsf{next}_T(q, \mathsf{calls}(f))) \in [0, 1]$$



Formula Trees

Simply And-Or trees holding the quantifier-free formulas that can be updated with concrete states to evaluate sub-formulas.

$$\forall^{S} q \in \text{changes}(a) : \quad q(a) \in [0, 20] \implies \text{duration}(\text{next}_{T}(q, \text{calls}(f))) \in [0, 1]$$

True

Instrumentation Problem

Need to pick points in the program to add instruments to produce the dynamic run

Could pick all points but this will be inefficient

Use the specification to decide where to instrument

Two phases

1. Identify the symbolic support of a quantification domain as the set of symbolic states that *could* produce a binding
2. Use the quantifier-free formula to find all necessary symbolic states forward reachable from the symbolic support

Minimal Instrumentation

Importantly, if we just consider these instrumentation points then we preserve verdicts.

Theorem

For SCFG(P), if $\mathcal{D}$ satisfies ϕ then the dynamic run produced by removing all states from $\mathcal{D}$ (by collapsing transitions) not identified as instrumentation points also satisfies ϕ.

Instrumentation is minimal with respect to reachability - but not necessarily with respect to other things e.g. dataflow

Optimisations

Generation Points
Now that we statically know when bindings can be created we can do a path-analysis to find all points where bindings are necessarily going to be extended and remove this iteration from the naive monitoring algorithm.

In other words, all binding-generation points can be identified completely statically.

Instrumentation for Indexing
We can also statically determine which bindings will be updated where, allowing us to store this information and use it to directly index the relevant formula trees. More information in TACAS tool paper.

The VyPR tool

Takes a Python program and a property specification file (written with our own specification-building library) and

1. Builds the SCFG
2. Identifies and adds relevant instrumentation points
3. Runs monitoring *asynchronously*
4. Outputs a verdict report once the program terminates

Experiments with VyPR

Monitor two properties on a sample (representative) program

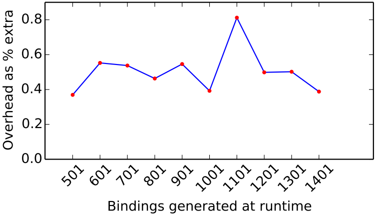
$$\forall^S q \in \text{changes}(a) : \\ q(a) \in [0, 80] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]$$

$$\forall^S q \in \text{changes}(a) : \ \forall^T t \in \ \text{future}(q, \text{calls}(f)) : \\ q(a) \in [0, 80] \implies \text{duration}(t) \in [0, 1]$$

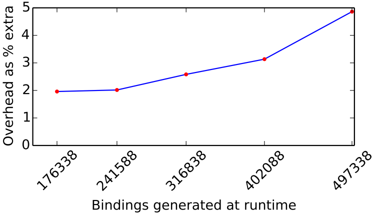
Questions

1. How much overhead does VyPR introduce?
2. How much does this depend on time between observations?

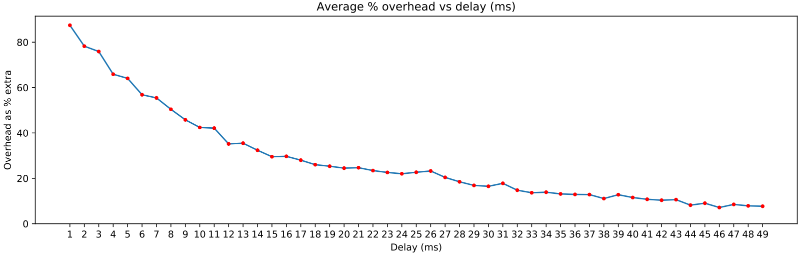
Results



Overhead as % extra
0.0
0.2
0.4
0.6
0.8
501
601
701
801
901
1001
1101
1201
1301
1401
Bindings generated at runtime



Overhead as % extra
0
1
2
3
4
5
176338
241588
316838
402088
497338
Bindings generated at runtime



Average % overhead vs delay (ms)
Overhead as % extra
0
20
40
60
80
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
Delay (ms)

Summary

New runtime verification framework for real-time temporal properties of Python functions

VyPR has now been extended to web services.

- The extension, along with its first major application to infrastructure at the CMS Experiment at CERN, is currently being presented at TACAS.

Future work

- Transformations on SCFG (and program) to reduce instrumentation
- Symbolic execution to reduce instrumentation
- Violation explanation