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 $\tau$ and a property we want the system to have $\phi$ 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 $\tau$, create a monitor from $\phi$ to observe $\tau$ and decide (at runtime) $\tau \in \mathcal{L}(\phi)$

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

Diagram:
- **property**
- **monitor**
  - **observe**
  - **feedback**
  - **verdict**
- **instrumentation**
  - **system**
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

□

\[
\left( \begin{array}{c}
\text{quick} \\
\left( \neg \text{proc } \mathcal{U}_{[0,5]} \text{ out} \right) \\
\land \Box_{10} \text{ fin}
\end{array} \right)
\]

\[
\begin{align*}
\text{quick} & \leftrightarrow \textbf{call} \ \text{process} \\
& \quad \textit{with} \ \text{quick} = 1 \\
\text{proc} & \leftrightarrow (\textbf{call} \ \text{rebalance}) \lor \\
& \quad (\textbf{call} \ \text{balanceIns}) \\
\text{out} & \leftrightarrow \textbf{call} \ \text{logging.log} \\
\text{fin} & \leftrightarrow \textbf{return} \ \text{process}
\end{align*}
\]
### Motivation 2

<table>
<thead>
<tr>
<th>Tool</th>
<th>Specification</th>
<th>Monitor</th>
<th>Deployment</th>
<th>Reaction</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aerial</td>
<td>none</td>
<td>dynamic programming</td>
<td>i</td>
<td>on</td>
<td>none</td>
</tr>
<tr>
<td>ARTIMon</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BeeepBeep</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DANA</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>detectEr</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E-ACSL</td>
<td>ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaMOP</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUnitRV</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Larva</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LogFire</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MarQ/</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QEA</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MonPoly</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mufin</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2U2</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RiTHM</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTC</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RV-Monitor</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STePr</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TempPsy/</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCLRCHECK</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VALOUR</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Motivation 2

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

HasNext [edit]

The Java 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.

∀ Iterator i. i.next() → ◻(i.hasNext() == 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 Enumeration does not get used when their underlying Vector is modified. The following parametric regular pattern can be used.
The Separation and Locality Problems

RV approaches often separate instrumentation and specification

- The requirement for an instrumentation mapping can mean that $\varphi$ 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.
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

\[
Program := x = expr | Program; Program |
\]
\[
\text{if} \ expr \ \text{then} \ Program \ (\text{else} \ Program) | |
\text{while} \ expr \ \text{do} \ Program | \text{for} \ expr \ \text{in} \ Program
\]

\[
expr := x | f(expr_1, \ldots, expr_n) | \text{arithExpr} | \text{boolExpr}
\]

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 \( \rightarrow \{ \text{changed, called, undefined} \} \)

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.
Constructing SCFG

We define a translation function recursively on the structure of programs. \( T \sigma, P \) gives the set of edges from symbolic state \( \sigma \) given the program \( P \).

For example, we translate an assignment as follows

\[
T(\sigma, x = expr; P) = \\
\{ \langle \sigma, \langle p(P), [x \mapsto \text{changed}] \rangle \rangle \} \cup T(\langle p(P), [x \mapsto \text{changed}] \rangle, P)
\]

if \( \text{fn}(expr) = \emptyset \), and

\[
\{ \langle \sigma, \langle p(P), [x_i \mapsto \text{changed}, f_i \mapsto \text{called}] \rangle \rangle \} \\
\cup T(\langle p(P), [x_i \mapsto \text{changed}, f_i \mapsto \text{called}] \rangle, P)
\]

for \( x_i \in \text{VarR} \) and \( f_i \in \text{fn}(expr) \) otherwise
A Notion of Traces: Dynamic Runs

We define dynamic runs over SCFG($P$) = $\langle V, E, \nu_s \rangle$

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 $\tau$ 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 $\sigma$ and $\sigma'$, 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.

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

\[\langle 0, \emptyset, \emptyset \rangle\]
\[\langle 0.1, [a \mapsto \text{changed}], [a \mapsto 10] \rangle\]
\[\langle 0.2, [i \mapsto \text{changed}], [a \mapsto 10, i \mapsto 0] \rangle\]
\[\langle 0.8, [f \mapsto \text{called}], [a \mapsto 10, i \mapsto 0] \rangle\]
\[\langle 0.9, [i \mapsto \text{changed}], [a \mapsto 10, i \mapsto 1] \rangle\]
\[\langle 2.1, [f \mapsto \text{called}], [a \mapsto 10, i \mapsto 1] \rangle\]
\[\langle 2.2, \emptyset, [a \mapsto 10] \rangle\]
\[\langle 2.3, [a \mapsto \text{changed}], [a \mapsto 20] \rangle\]
\[\langle 3.4, [f \mapsto \text{called}], [a \mapsto 20] \rangle\]
Control-Flow Temporal Logic

“The calls to function $f$ take less than 5 time units”

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

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

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

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

$$\forall^S q \in \text{changes}(x) : \forall^T t \in \text{future}_T(q, \text{calls}(f)) :$$

$$(q(x) \in (0, 5) \lor q(x) \in [0, 1]) \implies \text{duration}(t) \in (0, 10).$$
Control-Flow Temporal Logic

“The calls to function \( f \) take less than 5 time units”

\[ \forall^T t \in \text{calls}(f) : \text{duration}(t) \in (0, 5). \]

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

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

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

\[ \forall^S q \in \text{changes}(x) : \forall^T t \in \text{future}_T(q, \text{calls}(f)) : 
\quad (q(x) \in (0, 5) \lor q(x) \in [0, 1]) \implies \text{duration}(t) \in (0, 10). \]
Control-Flow Temporal Logic

“The calls to function \( f \) take less than 5 time units”

\[
\forall^T t \in \text{calls}(f) : \text{duration}(t) \in (0, 5).
\]

“Whenever \( x \) changes, its value remains unchanged until the next call of \( f \)” , i.e. \( f \) always sees every change to \( x \)

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

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

\[
\forall^S q \in \text{changes}(x) : \forall^T t \in \text{future}_T(q, \text{calls}(f)) : (q(x) \in (0, 5) \lor q(x) \in [0, 1]) \implies \text{duration}(t) \in (0, 10).
\]
Control-Flow Temporal Logic

“The calls to function $f$ take less than 5 time units”

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

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

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

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

$$\forall^S q \in \text{changes}(x) : \forall^T t \in \text{future}_T(q, \text{calls}(f)) :$$

$$(q(x) \in (0, 5) \lor q(x) \in [0, 1]) \implies \text{duration}(t) \in (0, 10).$$
Our Example

1. \(a = 10\)
2. for \(i\) in range(\(n\)):
   3. \(f(i)\)
4. \(a = 20\)
5. \(f(a)\)
6. \[
\begin{align*}
\forall^S q \in \text{changes}(a) : \\
q(a) \in [0, 20] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1]
\end{align*}
\]
Syntax

\[ \phi \quad ::= \quad \forall S q \in \Gamma_S : \phi \mid \forall T t \in \Gamma_T : \phi \mid \phi \lor \phi \mid \neg \phi \mid \phi_S \mid \phi_T \mid \text{true} \]

\[ \phi_S \quad ::= \quad S(x) = v \mid S(x) = S(x) \mid S(x) \in (n, m) \mid S(x) \in [n, m] \]

\[ \phi_T \quad ::= \quad \text{duration}(T) \in (n, m) \mid \text{duration}(T) \in [n, m] \]

\[ \Gamma_S \quad ::= \quad \text{changes}(x) \mid \text{future}_S(q, \text{changes}(x)) \mid \text{future}_S(t, \text{changes}(x)) \]

\[ \Gamma_T \quad ::= \quad \text{calls}(f) \mid \text{future}_T(q, \text{calls}(f)) \mid \text{future}_T(t, \text{calls}(f)) \]

\[ S \quad ::= \quad q \mid \text{source}(T) \mid \text{dest}(T) \mid \text{next}_S(S, \text{changes}(x)) \mid \text{next}_S(T, \text{changes}(x)) \]

\[ T \quad ::= \quad t \mid \text{incident}(S) \mid \text{next}_T(S, \text{calls}(f)) \mid \text{next}_T(T, \text{calls}(f)) \]

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

\[
\phi := \forall S \ q \in \Gamma_S : \phi \ | \ \forall T \ t \in \Gamma_T : \phi \ | \ \phi \lor \phi \ | \ \neg \phi \ | \ \phi_S \ | \ \phi_T \ | \ true
\]

\[
\phi_S := S(x) = v \ | \ S(x) = S(x) \ | \ S(x) \in (n, m) \ | \ S(x) \in [n, m]
\]

\[
\phi_T := \text{duration}(T) \in (n, m) \ | \ \text{duration}(T) \in [n, m]
\]

\[
\Gamma_S := \text{changes}(x) \ | \ \text{future}_S(q, \text{changes}(x)) \ | \ \text{future}_S(t, \text{changes}(x))
\]

\[
\Gamma_T := \text{calls}(f) \ | \ \text{future}_T(q, \text{calls}(f)) \ | \ \text{future}_T(t, \text{calls}(f))
\]

\[
S := q \ | \ \text{source}(T) \ | \ \text{dest}(T) \ | \ \text{next}_S(S, \text{changes}(x)) \ | \ \text{next}_S(T, \text{changes}(x))
\]

\[
T := t \ | \ \text{incident}(S) \ | \ \text{next}_T(S, \text{calls}(f)) \ | \ \text{next}_T(T, \text{calls}(f))
\]

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

\[ \phi \quad ::= \quad \forall S q \in \Gamma_S : \phi \mid \forall T t \in \Gamma_T : \phi \mid \phi \lor \phi \mid \neg \phi \mid \phi_S \mid \phi_T \mid \text{true} \]

\[ \phi_S \quad ::= \quad S(x) = v \mid S(x) = S(x) \mid S(x) \in (n, m) \mid S(x) \in [n, m] \]

\[ \phi_T \quad ::= \quad \text{duration}(T) \in (n, m) \mid \text{duration}(T) \in [n, m] \]

\[ \Gamma_S \quad ::= \quad \text{changes}(x) \mid \text{future}_S(q, \text{changes}(x)) \mid \text{future}_S(t, \text{changes}(x)) \]

\[ \Gamma_T \quad ::= \quad \text{calls}(f) \mid \text{future}_T(q, \text{calls}(f)) \mid \text{future}_T(t, \text{calls}(f)) \]

\[ S \quad ::= \quad q \mid \text{source}(T) \mid \text{dest}(T) \mid \text{next}_S(S, \text{changes}(x)) \mid \text{next}_S(T, \text{changes}(x)) \]

\[ T \quad ::= \quad t \mid \text{incident}(S) \mid \text{next}_T(S, \text{calls}(f)) \mid \text{next}_T(T, \text{calls}(f)) \]

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 $\Gamma$ if $\mathcal{D}, q \vdash \Gamma$ (or $\mathcal{D}, tr \vdash \Gamma$)

$\mathcal{D}, \langle t, \sigma, \tau \rangle \vdash \text{changes}(x)$ iff $\sigma(x) = \text{changed}$

$\mathcal{D}, q \vdash \text{future}_S(s, \text{changes}(x))$ iff $t(q) > t(s)$ and $\mathcal{D}, q \vdash \text{changes}(x)$

$\mathcal{D}, tr \vdash \text{calls}(f)$ iff

for every path $\pi \in \text{paths}(tr)$ there is:

some $\langle \sigma_1, \sigma_2 \rangle \in \pi$

such that $\sigma_2(f) = \text{called}$

$\mathcal{D}, tr \vdash \text{future}_T(s, \text{calls}(f))$ iff $t(tr) > t(s)$ and $\mathcal{D}, tr \vdash \text{calls}(f)$

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 $D$ satisfies point of interest $\Gamma$ if $D, q \vdash \Gamma$ (or $D, tr \vdash \Gamma$)
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 $\Gamma_S$ (and $\Gamma_T$) to also stand for this set.

This could be computed by iterating over $\mathcal{D}$ and checking $\vdash \Gamma$ for each state (or transition).
where we evaluate quantifier-free formulas on the dynamic run with respect to a given binding.
Semantics

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

Where we evaluate quantifier-free formulas on the dynamic run with respect to a given binding.
Evaluating non-temporal formulas relatively straightforward given some functions operating on states and transitions e.g.

\[ \text{source}(\langle q_1, q_2 \rangle) = q_1. \]

\[
\begin{align*}
\text{eval}(\mathcal{D}, \beta, q) &= \beta(q) \\
\text{eval}(\mathcal{D}, \beta, tr) &= \beta(tr) \\
\text{eval}(\mathcal{D}, \beta, \text{source}(T)) &= \text{source}(\text{eval}(\mathcal{D}, \beta, T)) \\
\text{eval}(\mathcal{D}, \beta, \text{dest}(T)) &= \text{dest}(\text{eval}(\mathcal{D}, \beta, T)) \\
\text{eval}(\mathcal{D}, \beta, \text{incident}(S)) &= \text{incident}(\mathcal{D}, \text{eval}(\mathcal{D}, \beta, S))
\end{align*}
\]
Evaluating Temporal Formulas

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

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

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

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

\[ t(tr) > t(\text{eval}(D, \beta, X)) \text{ and } D, tr \vdash \text{calls}(f) \text{ and there is no } tr' \text{ with } t(\text{eval}(D, \beta, X)) < t(tr') < t(tr) \text{ and } D, tr' \vdash \text{calls}(f) \]
Finally, a dynamic run $\mathcal{D}$ satisfies a (well-formed, well-defined) CFTL formula $\phi$ if $\mathcal{D}, [] \models \phi$, otherwise $\mathcal{D}$ violates $\phi$. 
Our Example

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

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

\[
\begin{align*}
\langle 0, [], [] \rangle & \\
\langle 0.1, [a \mapsto \text{changed}], [a \mapsto 10] \rangle & \in \text{changes}(a) \\
\langle 0.2, [i \mapsto \text{changed}], [a \mapsto 10, i \mapsto 0] \rangle & \\
\langle 0.8, [f \mapsto \text{called}], [a \mapsto 10, i \mapsto 0] \rangle & \in \text{calls}(f) \\
\langle 0.9, [i \mapsto \text{changed}], [a \mapsto 10, i \mapsto 1] \rangle & \\
\langle 2.1, [f \mapsto \text{called}], [a \mapsto 10, i \mapsto 1] \rangle & \in \text{calls}(f) \\
\langle 2.2, [], [a \mapsto 10] \rangle & \\
\langle 2.3, [a \mapsto \text{changed}], [a \mapsto 20] \rangle & \in \text{changes}(a) \\
\langle 3.4, [f \mapsto \text{called}], [a \mapsto 20] \rangle & \in \text{calls}(f)
\end{align*}
\]

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

\[
\langle 0, [], [] \rangle \\
\langle 0.1, [a \mapsto \text{changed}], [a \mapsto 10] \rangle \quad \in \text{changes}(a) \\
\langle 0.2, [i \mapsto \text{changed}], [a \mapsto 10, i \mapsto 0] \rangle \\
\langle 0.8, [f \mapsto \text{called}], [a \mapsto 10, i \mapsto 0] \rangle \quad \in \text{calls}(f) \\
\langle 0.9, [i \mapsto \text{changed}], [a \mapsto 10, i \mapsto 1] \rangle \\
\langle 2.1, [f \mapsto \text{called}], [a \mapsto 10, i \mapsto 1] \rangle \quad \in \text{calls}(f) \\
\langle 2.2, [], [a \mapsto 10] \rangle \\
\langle 2.3, [a \mapsto \text{changed}], [a \mapsto 20] \rangle \quad \in \text{changes}(a) \\
\langle 3.4, [f \mapsto \text{called}], [a \mapsto 20] \rangle \quad \in \text{calls}(f)
\]

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

\[
\langle 0, [], [] \rangle \\
\langle 0.1, [a \mapsto \text{changed}], [a \mapsto 10] \rangle \quad \in \text{changes}(a) \\
\langle 0.2, [i \mapsto \text{changed}], [a \mapsto 10, i \mapsto 0] \rangle \\
\langle 0.8, [f \mapsto \text{called}], [a \mapsto 10, i \mapsto 0] \rangle \quad \in \text{calls}(f) \\
\langle 0.9, [i \mapsto \text{changed}], [a \mapsto 10, i \mapsto 1] \rangle \\
\langle 2.1, [f \mapsto \text{called}], [a \mapsto 10, i \mapsto 1] \rangle \quad \in \text{calls}(f) \\
\langle 2.2, [], [a \mapsto 10] \rangle \\
\langle 2.3, [a \mapsto \text{changed}], [a \mapsto 20] \rangle \quad \in \text{changes}(a) \\
\langle 3.4, [f \mapsto \text{called}], [a \mapsto 20] \rangle \quad \in \text{calls}(f)
\]

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

\langle 0, []_1, [] \rangle
\langle 0.1, [a \mapsto \text{changed}]_2, [a \mapsto 10] \rangle \quad \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 \quad \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 \quad \in \text{calls}(f)
\langle 2.2, []_4, [a \mapsto 10] \rangle
\langle 2.3, [a \mapsto \text{changed}]_5, [a \mapsto 20] \rangle \quad \in \text{changes}(a)
\langle 3.4, [f \mapsto \text{called}]_6, [a \mapsto 20] \rangle \quad \in \text{calls}(f)

\forall^S q \in \text{changes}(a) :
q(a) \in [0, 20] \quad \Longrightarrow \quad \text{duration}(next_T(q, \text{calls}(f))) \in [0, 1]
Our Example

\begin{align*}
\langle 0, [], [] \rangle \\
\langle 0.1, [a \mapsto \text{changed}], [a \mapsto 10] \rangle &\in \text{changes}(a) \\
\langle 0.2, [i \mapsto \text{changed}], [a \mapsto 10, i \mapsto 0] \rangle \\
\langle 0.8, [f \mapsto \text{called}], [a \mapsto 10, i \mapsto 0] \rangle &\in \text{calls}(f) \\
\langle 0.9, [i \mapsto \text{changed}], [a \mapsto 10, i \mapsto 1] \rangle \\
\langle 2.1, [f \mapsto \text{called}], [a \mapsto 10, i \mapsto 1] \rangle &\in \text{calls}(f) \\
\langle 2.2, [], [a \mapsto 10] \rangle \\
\langle 2.3, [a \mapsto \text{changed}], [a \mapsto 20] \rangle &\in \text{changes}(a) \\
\langle 3.4, [f \mapsto \text{called}], [a \mapsto 20] \rangle &\in \text{calls}(f)
\end{align*}

\[ \forall S q \in \text{changes}(a) : \]
\[ q(a) \in [0, 20] \implies \text{duration}(\text{next}_T(q, \text{calls}(f))) \in [0, 1] \]
**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 $\Gamma_1$ then create a new binding
2. If there is a binding $\beta$ that can be extended by $q$ for a quantification domain $\Gamma_i$ 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.

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

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

\[ \forall S \ q \in \text{changes}(a) : q(a) \in [0, 20] \implies \text{duration(}next_T(q, \text{calls}(f))\text{)} \in [0, 1] \]
Formula Trees

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

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

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

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

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

**Theorem**

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

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 \textit{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

Slide 40
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