Explaining Violations of Properties in Control-Flow Temporal Logic

Joshua Heneage Dawes University of Manchester, Manchester, UK
CERN, Geneva, Switzerland
joshua.dawes@cern.ch

Giles Reger University of Manchester, Manchester, UK

Runtime Verification 2019, Porto, Portugal

Outline

Explanation in RV

Overview of Control-Flow Temporal Logic

An Extension of the Semantics

Severity of Violations

Path Reconstruction and Comparison

Explanation with VyPR and applications at CERN

Difficulties in Explanation Research

Given a failure wrt a specification, explanation is a natural next step.

Definition of “explanation” depends on the domain and the specific program being monitored.

In some cases, program variable recording could suffice.

In other cases, comparison of deviations in control-flow might be better.

General problem

Runtime Verification ideally is an automatic process (at least with VyPR, our framework at CERN), once specifications exist.

Therefore, explanation should be too.

That is to say: any explanation mechanism must be able to give a developer useful feedback about what could be causing failure, preferably without intermediate input.

A Low-level Specification Language

The explanation approach to be presented is independent of specification language.

Here we use Control-Flow Temporal Logic (CFTL), as used in the environment we work with at CERN.

So a quick introduction.

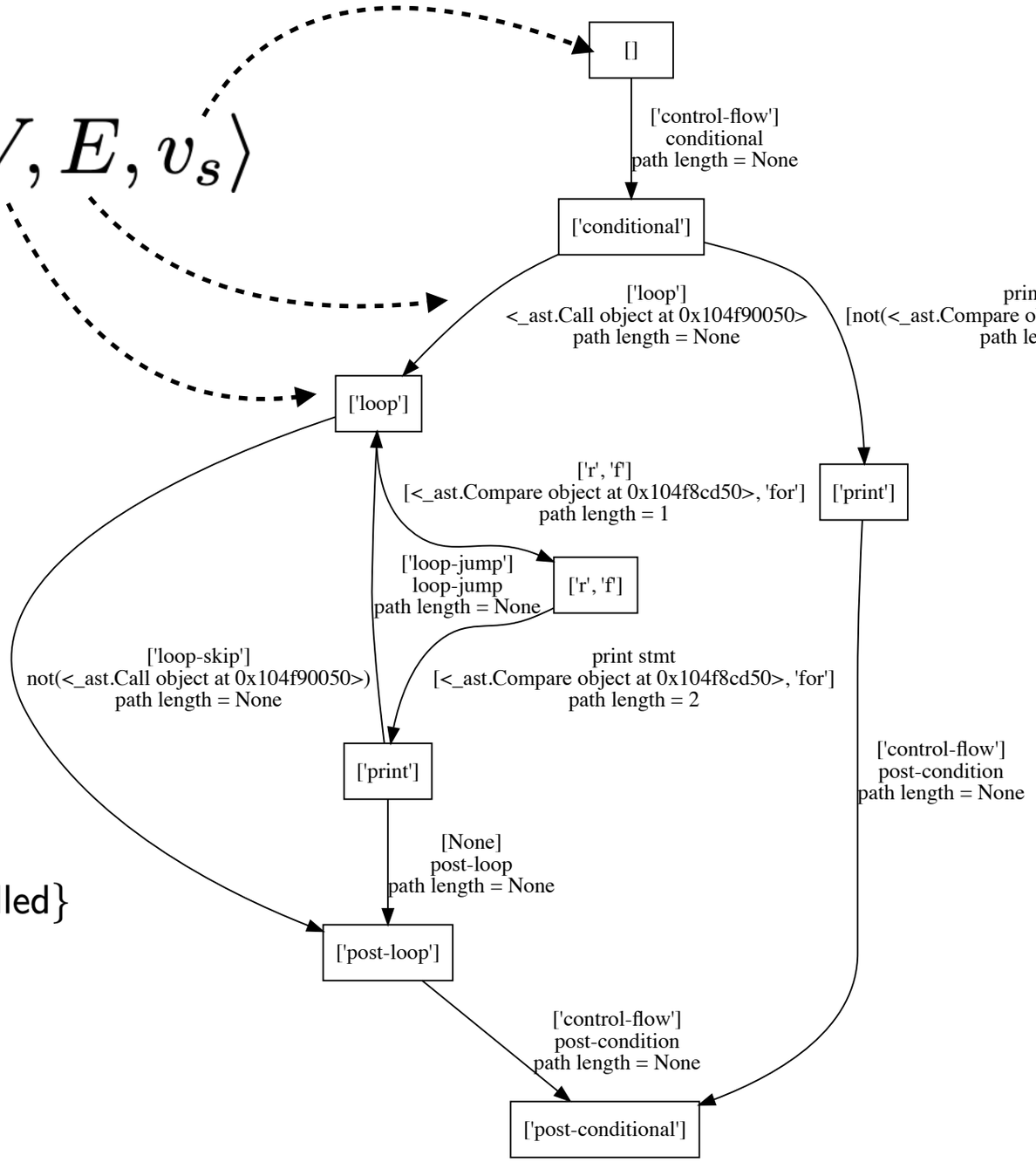
A static model of programs: *Symbolic Control-Flow Graphs (SCFGs)*

$$\mathsf{SCFG}(P) = \langle V, E, v_s \rangle$$

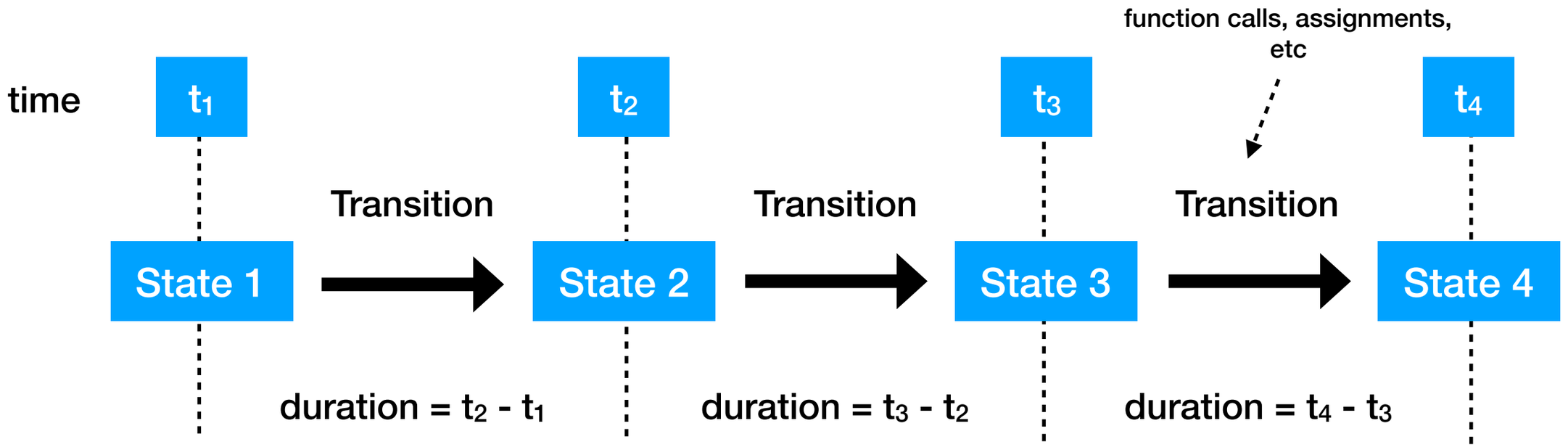
```
if n > 1:
  for i in range(n):
    r = f(i)
    print(r)
else:
  print("nope")
```

variables/functions in P

$\sigma : \mathsf{Sym} \to \{\mathsf{undefined}, \mathsf{unchanged}, \mathsf{changed}, \mathsf{called}\}$



Dynamic Runs

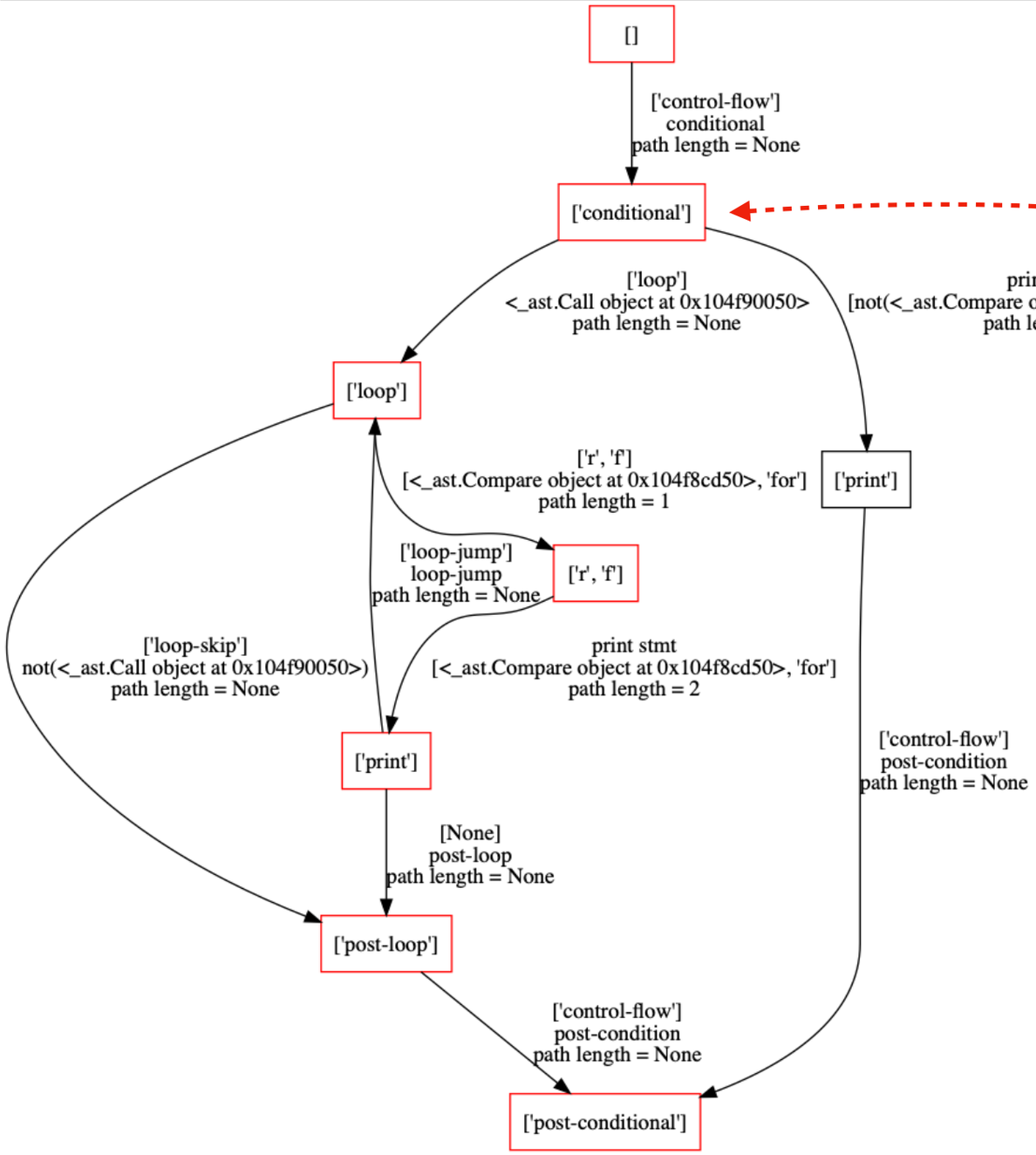


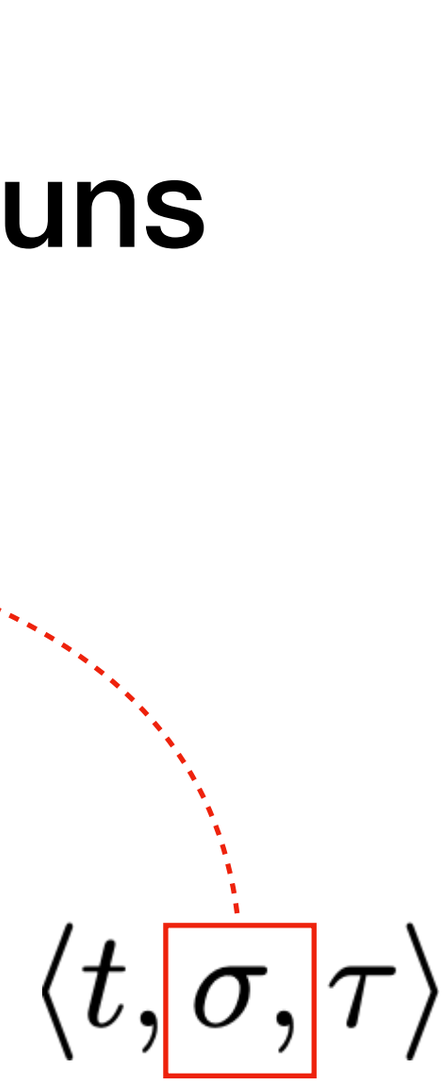
We think of program runs as **dynamic runs**:

Concrete States (instantaneous checkpoints with an associated timestamp) - $\langle t, \sigma, \tau \rangle$

Transitions (computation performed to reach one state from another) - pairs of states - $\langle \langle t, \sigma, \tau \rangle, \langle t', \sigma', \tau' \rangle \rangle$

SCFGs to Dynamic Runs





Dynamic Runs identify with paths through SCFGs.

Each concrete state in a Dynamic Run contains a symbolic state.

CFTL Example

CFTL formulas reason about states and transitions from dynamic runs.

$$\forall q \in \mathsf{fromState}(\sigma) :$$
$$(q(\texttt{category}) = \texttt{A} \wedge q(\texttt{db}) = \texttt{oracle}) \implies$$
$$\mathsf{length}(\mathsf{result}(\mathsf{next}(q, \mathsf{calls}(\texttt{query})))(\texttt{rows})) < 10$$

"For each concrete state we get from the symbolic state *sigma*, if the category is *A* and the database is *oracle*, then the length of the list of results returned from the query is less than 10."

Points of Failure in Dynamic Runs

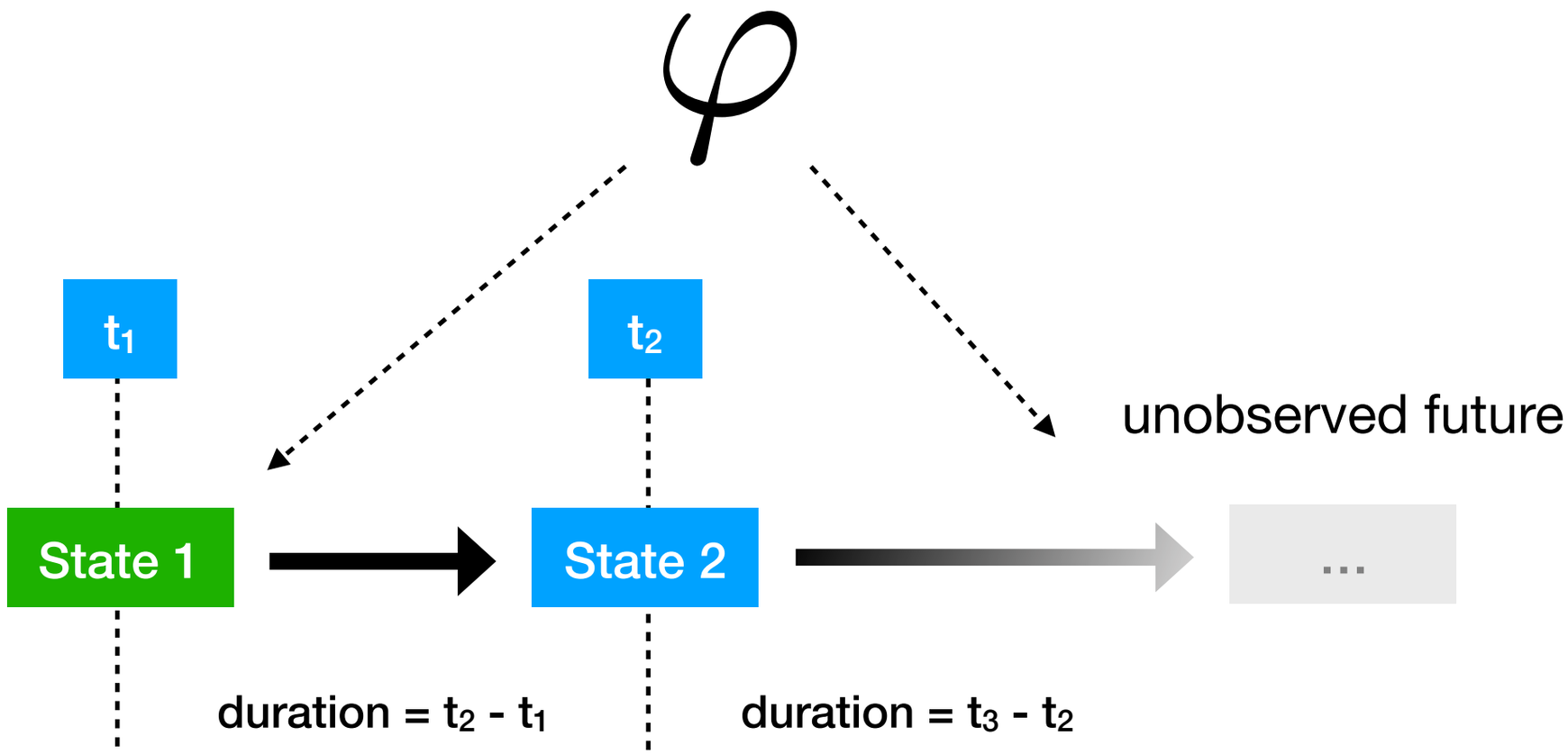
Standard CFTL semantics is defined over total dynamic runs and has truth domain $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$

Failure requires a notion of permanent change of verdict.

In other words, impartiality.

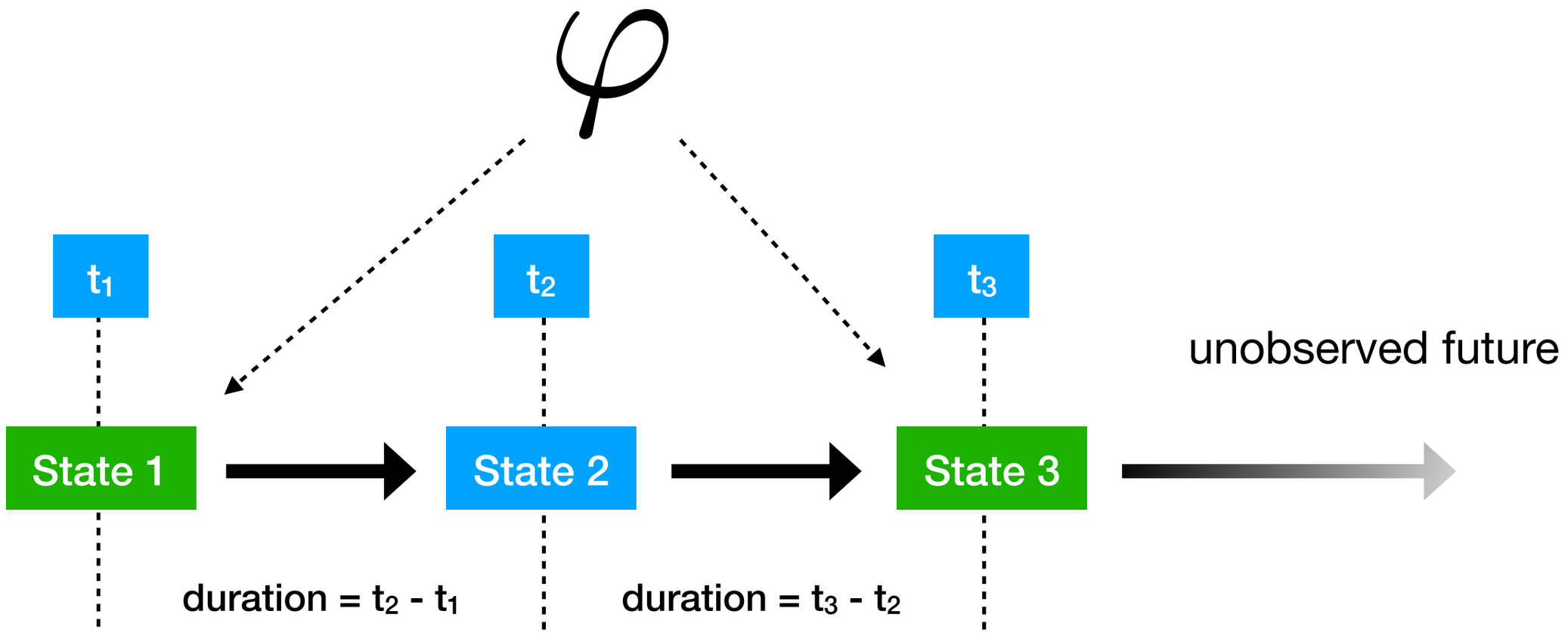
…a point of failure can’t exist if it’s possible for a property to be satisfied in the future!

A truth domain for partial dynamic runs



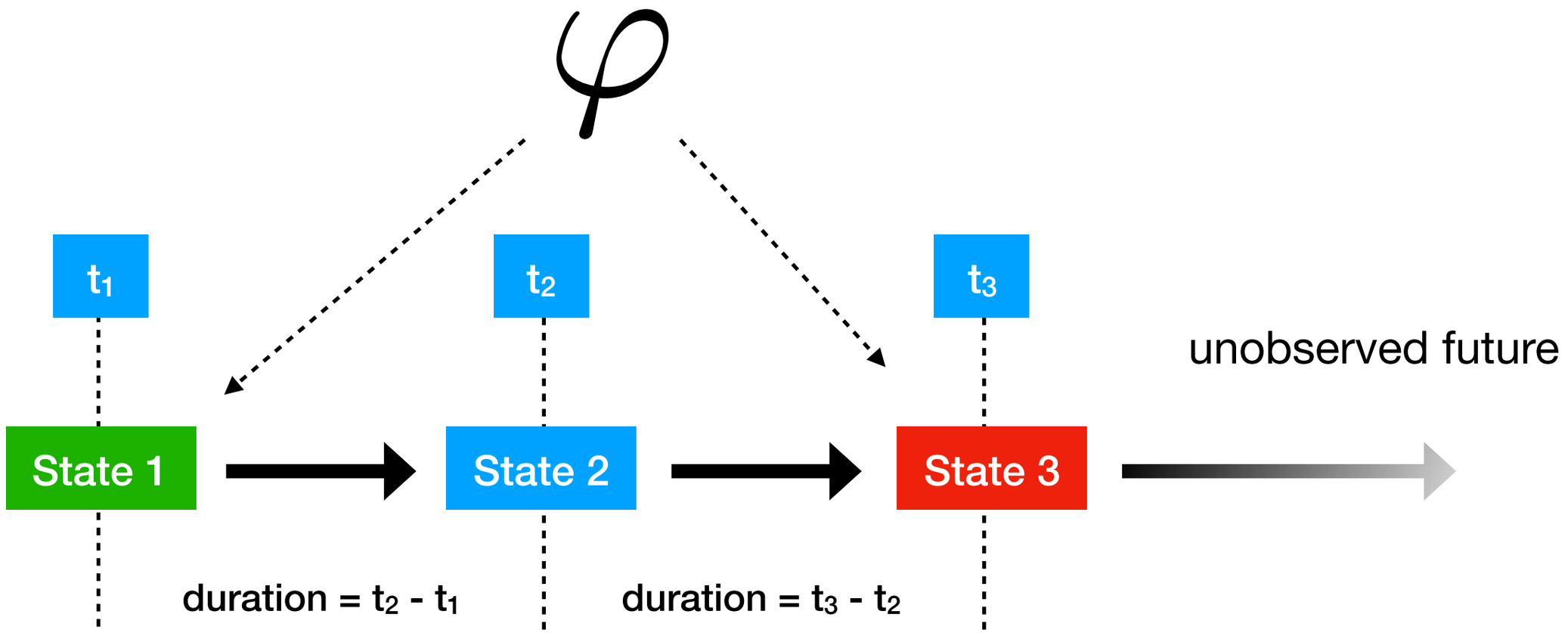
notSure

A truth domain for partial dynamic runs



trueSoFar

A truth domain for partial dynamic runs



false

(same as total semantics)

A truth domain for partial dynamic runs

$$\mathsf{false} < \mathsf{notSure} < \mathsf{trueSoFar}$$

$$\bigsqcup \equiv \bigvee \qquad \bigsqcap \equiv \bigwedge$$

$$\mathsf{false} \wedge \mathsf{notSure} \equiv \mathsf{false}$$

$$\mathsf{false} \vee \mathsf{notSure} \equiv \mathsf{notSure}$$

A 3-valued CFTL semantics

The semantics presented at SAC 2019 used an eval function.

Dynamic Run Binding from Quantification

$$\mathsf{eval}(\mathcal{D}, \theta, X)$$

State or Transition expression

Our first step is to modify this for partial dynamic runs to return null if an observation is not found.

Possibly partial binding

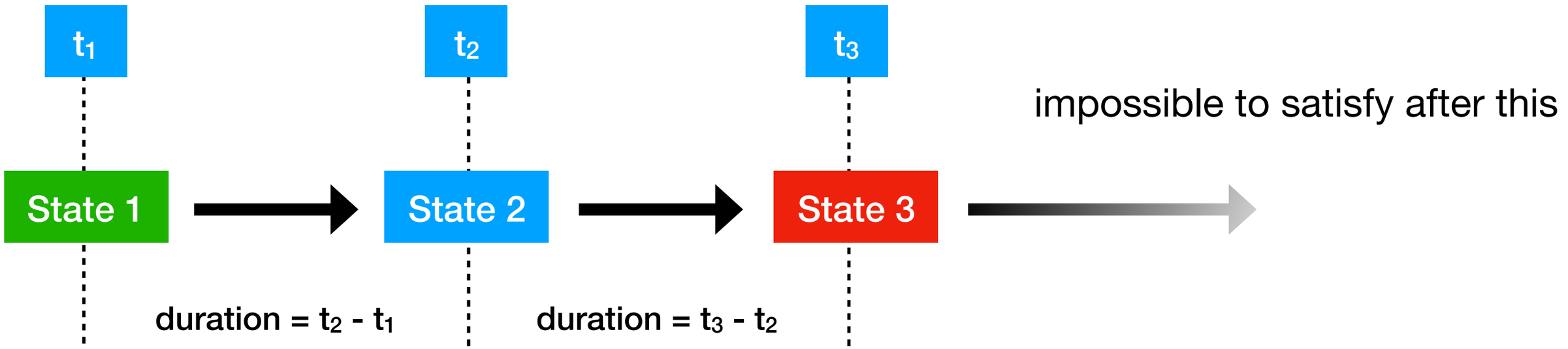
Partial Dynamic Run

$$\mathsf{eval}_p(\mathcal{D}_p, \theta, X)$$

If a part of a formula is given null by eval and we cannot reach a verdict otherwise, we conclude notSure. Otherwise we can conclude false or trueSoFar.

Falsifying Observations

The falsifying observation for a failing dynamic run is the first observation for which the verdict was false, before which the verdict was not false.



State 3 is a falsifying observation

Quantitative Semantics

Given the notion of a falsifying observation, it's natural to ask by how much it was falsifying.

Key facts:

- There is (part of) an atom for which an observation that is falsifying exists.

$$\forall q \in \mathsf{changes}(\texttt{limit}) :$$
$$\boxed{\mathsf{duration}(\mathsf{next}(q, \mathsf{calls}(\texttt{query})))} < \boxed{q(\texttt{limit})}$$

- Based on the atom, we can define a metric over failure (we give only one case here):

$$\text{With } \alpha = (\mathsf{duration}(t) \in I),$$
$$\mathsf{Sev}(\alpha, c) = \inf\{|\mathsf{duration}(c) - n| : n \in I\} \cdot \mathcal{X}(\alpha, c)$$
$$\text{for } \mathcal{X}(\alpha, c) = 1 \text{ if } \mathsf{duration}(c) \in I, -1$$

Complementary Explanation Approaches

When monitoring for CFTL specifications, we have immediate access to program state (variable values) and control flow.

Thanks to CFTL's low level of abstraction.

Hence an explanation approach can consist of comparing program state and control flow across multiple runs.

Comparing Program Paths

Instrumenting only certain program points to check a specification means we often cannot decide a path based just on these points.

Branch-aware Dynamic Run

Dynamic runs are made *branch-aware* by additional instrumentation at branching points.

A path as a sequence of edges through the SCFG

π

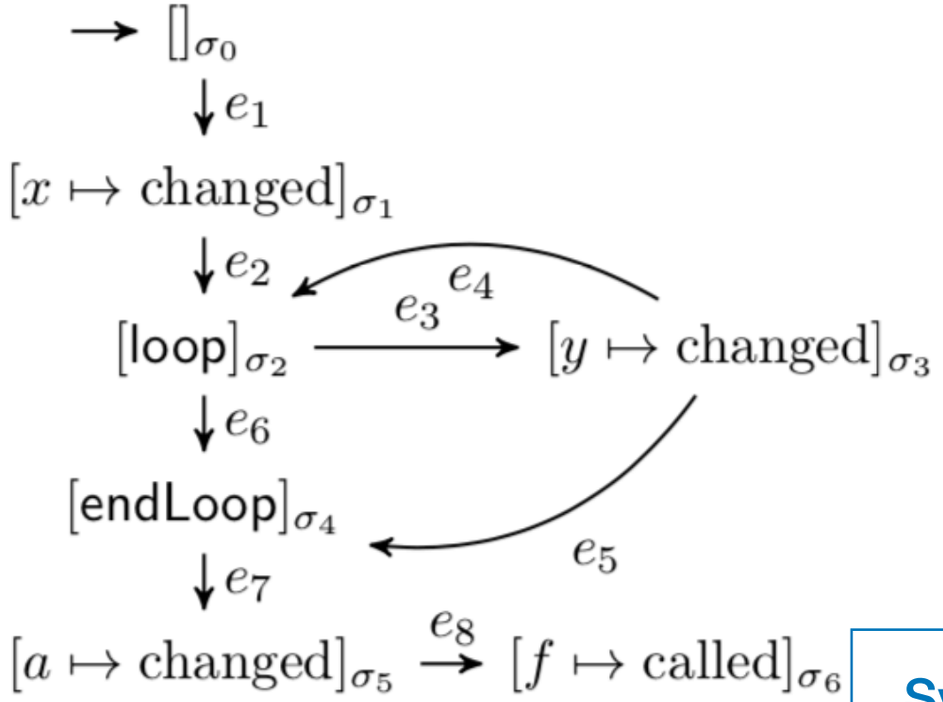
A parse tree representing derivation of the path via the context-free grammar

SCFG → Context-free grammar → $T(\pi)$

SCFG to CFG transformation

Doing this for an arbitrary SCFG is a matter of defining the CFG for each component and deriving the *composite* CFG *bottom-up*.

We define the components in such a way that the CFG is unambiguous.



$\vec{\sigma_0}$ →
$\vec{\sigma_1}$ → e_2 $\vec{\sigma_2}$ $\vec{\sigma_4}$
$\vec{\sigma_2}$ → e_3 $\vec{\sigma_3}$ | e_6
$\vec{\sigma_3}$ → e_4 $\vec{\sigma_2}$ | e_5
$\vec{\sigma_4}$ → e_7 $\vec{\sigma_5}$
$\vec{\sigma_5}$ → e_8 $\vec{\sigma_6}$
$\vec{\sigma_6}$ → ϵ

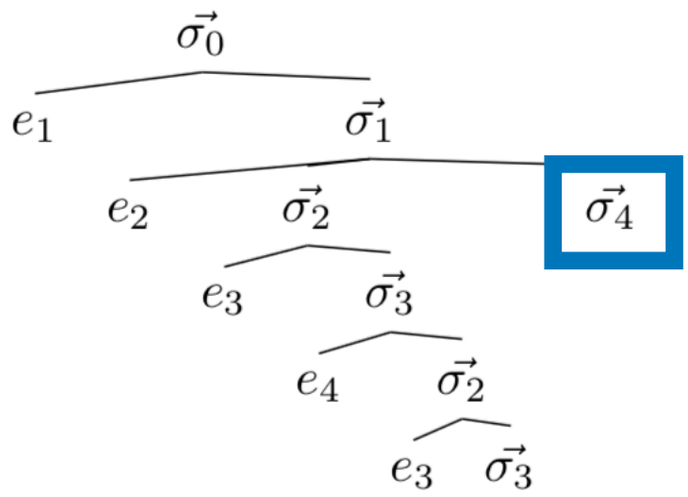
Paths through SCFGs are sequences of edges, so terminal symbols are edges.

Symbols corresponding to symbolic states are **non-terminal.**

Parse Trees

With unambiguous CFGs, parse tree derivation is trivial with a 1-step lookahead.

$T(\pi)$



Paths leading to observations are not complete, so the right-hand-sides of their parse trees have leaves with non-terminal symbols.

$$\pi_1, \ldots, \pi_n$$

SCFG to CFG to Parse Tree

$$T(\pi_1), \ldots, T(\pi_n)$$

Parse Tree Intersection

$$T(\pi_1) \cap \cdots \cap T(\pi_n)$$

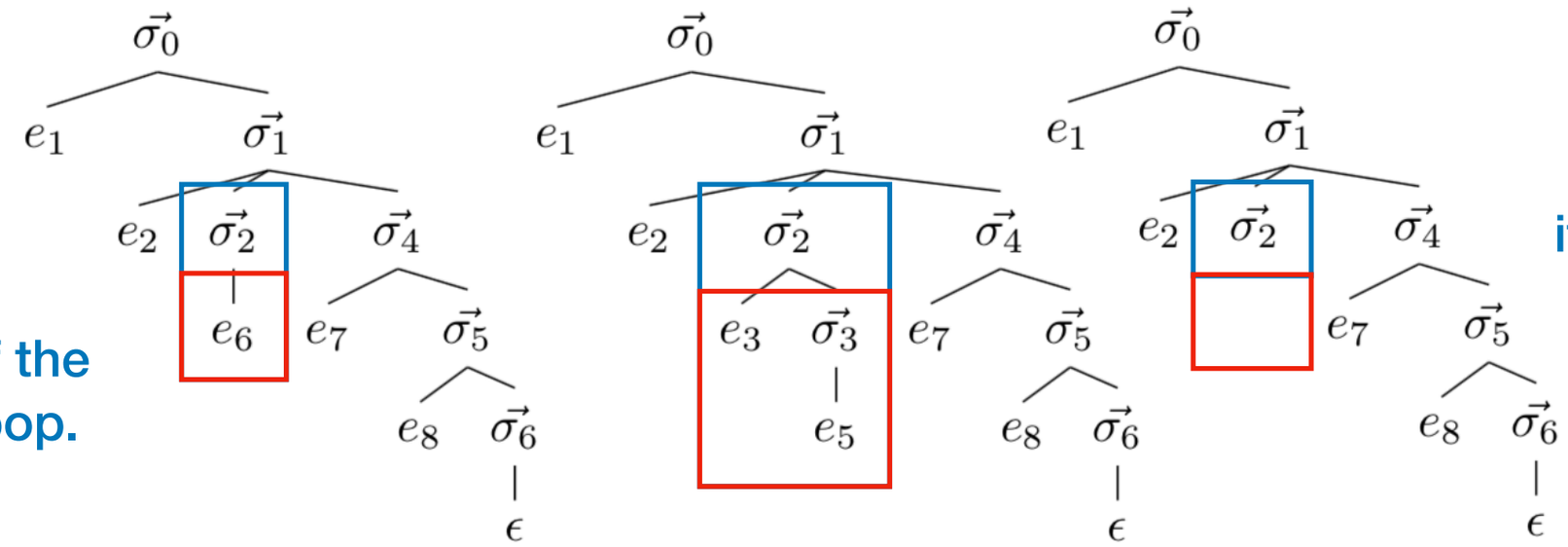
We define *intersection* to give
insight into branching and
loop iteration differences

Parse Tree Intersection

Our paper gives a recursive definition.

In general, the intersection of two parse trees is the parse tree containing all subtrees that are common up to the path with which one can reach them.

$\pi_1 = e_1\ e_2\ \underline{e_6}\ e_7\ e_8$ $\pi_2 = e_1\ e_2\ \underline{e_3\ e_5}\ e_7\ e_8$ $\pi_1 \cap \pi_2 = e_1\ e_2\ \vec{\sigma_2}\ e_7\ e_8$



This part of the path is a loop.

Intersection shows that iterations vary.

Comparison

By combining this path comparison approach with information about verdicts, we can determine whether certain program paths are problematic.

Current research is looking at automating the discovery of problematic control flow as much as possible.

Explanation with VyPR

VyPR

GPLv3

VyPR - a framework under active development at the CMS Experiment at CERN for automated performance analysis of Python programs using RV.

http://cern.ch/vypr and http://github.com/pyvypr

Path and state comparison accessible to developers via an analysis library.

Path Comparison with VyPR's Analysis Library

```
import VyPRAnalysis as analysis
import VyPRAnalysis.orm.operations as ops

analysis.set_config_file("VyPRAnalysis/config.json")
```

Connect to a verdict server

```
functions = analysis.list_functions()
f = functions[0]
```

Fix a function and a property over that function

```
verdicts = f.get_verdicts()
observations = [
  verdicts[0].get_observations()[0],
  verdicts[1].get_observations()[0]
]
obs_collection = ops.ObservationCollection(
  observations
)
```

Get a list of observations that were required to reach each verdict

```
path_collection = obs_collection.to_paths()
path_collection.show_critical_points_in_file(
  filename="critical_points"
)
```

Determine the points in control-flow at which paths leading to those observations diverged.

Output

```
@app.route('/paths_branching_test/<int:n>/', methods=["GET", "POST"])
def paths_branching_test(n):
  a = 20; c = 10;
*  if n > 10:
    l = []
    for i in range(n):
      print("test1")
      l.append(i**2)
      x = 15
      if x > 10:
        print("test 2")
else:
  l = []
f(l)
```

critical points in control flow are currently indicated with a star.

To conclude…

We have introduced some of the first work in the direction of explanation in RV.

Applying our approach in the context of CFTL gives a way to explain performance drops and low-level behaviour.

The techniques here have been applied to CMS infrastructure and have revealed insight into 1) behaviour of the code and even 2) behaviour of users.

Development is very much active at the CMS Experiment at CERN.