Static Slicing: A First Step Towards Explanation of CFTL Failures

Joshua Dawes[1,2,3]

[1]University of Manchester, Manchester, UK

[2]CERN, Geneva, Switzerland

[3]joshua.dawes@cern.ch

Summary so far

Technical report - preprint on arXiv - “Specification of State and Time Constraints for Runtime Verification of Functions” - detailed description of CFTL, instrumentation and monitoring.

Theory paper accepted at SAC-SVT 2019 - “Specification of Temporal Properties of Functions for Runtime Verification” - paper outlining CFTL, monitoring and small experimental data.

Applied paper accepted at TACAS 2019 with certification of software quality - “VyPR2: A Framework for Runtime Verification of Python Web Services” - paper outlining VyPR, its application to the web service setting, and its use in finding performance drops in the Conditions Uploader.

VyPR website `http://cern.ch/vypr`.

Contents of this talk

Static Slicing as an Explanation Strategy

CFTL constraints are over state and time.

Often, we care about time taken with respect to existing state.

In normal control-flow, if a function is called, one major thing to which we have access is the variables in the control-flow that might contribute to arguments of the function called.

`f(a, b)` - what contributes to the computation of `a` and `b`?

It seems natural to pair this notion of affecting variables with CFTL specifications.

What constitutes an explanation?

Given the work so far, explanation can mean a few things:

For which initial variable values has f failed my constraint?

For a given failure threshold, which initial variable values have caused this?

During which times did a constraint fail, and what were the values of the initial variables during those times?

Basically, explanation-by-analysis.

Static slicing

Program slicing was first proposed in 1981 as a way to help engineers find which parts of their programs are relevant to a specific statement.

We need a form of this; finding the variables that are relevant to computation of some other variable in the future.

Since we have to instrument to take only the data that we need, we have to do this statically to determine where to take data during runtime.

In the remainder of this talk, I give introductory material on static slicing and highlight problems still to be addressed. For example, we currently only deal with primitive types, and certain types of dependencies (generated by statements other than assignments) are not handled yet.

Initial Variables

```
a = args[0]
b = args[1]
e = a + b
r = f(e)
```

What are the initial variables on which e depends?

$e \mapsto \{a, b\} \mapsto \{args[0], args[1]\}$ - this can be thought of as a graph.

What if there is branching? The structure becomes more complex.

We have to relate dependence to reachability in control-flow.

Solution? Use the Symbolic Control-Flow Graph.

A Modified Symbolic Control-Flow Graph

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

V contains vertices, which are states showing statuses of program variables/functions.

E contains edges, which represent transitions between statuses.

We augment edges with assignment tuples

$$\langle t, \{x_1, \ldots, x_n\}, \{h_1, \ldots, h_n\} \rangle$$

t is the target of the assignment, x_i are normal variables that have been defined by a previous assignment statement and h_i are inherited variables that are defined implicitly, usually by a loop as a loop counter. Assignment to inherited variables removes their inherited status.

Example

```
sum = 0
for i in range(10):
  i = i + 1
  sum = sum + i**2
```

`sum` is normal and `i` is inherited in `i = i + 1`, but normal in `sum = sum + i**2`.

The type of a variable found in an expression affects how we instrument for its value.

For normal variables, we wait until we find the definition.

For inherited variables, we instrument just before the statement that uses the inherited variable.

Determining Initial Variables - theory revisited

Atom-driven instrumentation instruments for a CFTL formula $\varphi \equiv \forall q_1 \in \Gamma_1 : \cdots : \forall q_n \in \Gamma_n : \psi(q_1, \ldots, q_n)$ by using the formula structure.

It constructs the Binding Space $\mathcal{B}_\varphi$ of φ, ie, the set of points in code that could generate a change we care about.

Then, for each $\beta \in \mathcal{B}_\varphi$, it uses the set of atoms in φ, A_φ, to traverse the SCFG, finding the vertices/edges around β that are relevant.

So the set of instrumentation points can be partitioned to form a tree, which we denote by $\mathcal{H}_\varphi$.

We write $\mathcal{H}_\varphi(\beta, \alpha)$ to denote the instrumentation points required for the atom α based on the binding β.

Integrating static slicing into this theory

For a given $\beta \in \mathcal{B}_\varphi$, consider $\mathcal{H}_\varphi(\beta, \alpha)$.

Fix $e \in \mathcal{H}_\varphi(\beta, \alpha)$ such that e is an edge in the SCFG representing a call to the function f.

Let $\text{params}(f)$ be the set of parameters required by f, and let $\text{initial}(e, x)$ be the set of vertices in the SCFG whose incident edges are assignments to initial variables wrt x.

Then, for such an e, we care about

$$\bigcup_{x \in \text{params}(f)} \text{initial}(e, x)$$

We will now see the outline of an algorithm for determining initial variables based on primitive types. I haven't addressed reference types yet - this will be a significant part of the work in making this approach useful.

Finding Initial Variables of Primitive Types

Let e be an edge representing a call to the function f and let inherited(e) be the set of variables inherited in the scope in which we find e. We first compute params(f) (as well as we can do statically).

Let required $=$ params(f)$\backslash$inherited(e) (parameters of f without inherited variables).

Let incident(source(e)) denote the set of incident edges of the source of e.

Recurse on each of $e' \in$ incident(source(e)), but instead:

For each e', replace params(f) with the set of variables used on e' and remove inherited variables. Then add the result to required. Add target(e') to the final list of required variables if its assignment target is a required variable.

Potential Variables

The algorithm on the previous slide determines $\bigcup_{x \in \mathrm{params}(f)} \mathrm{initial}(e, x)$ for some e.

It does so by recursing backwards through the SCFG on every possible branch.

More formally, let Π be the set of paths in the SCFG from the edge e back to the starting vertex v_s.

Each $\pi \in \Pi$ may admit its own subset of $\mathrm{initial}(e, x)$ for each variable x. Denote this by $\mathrm{initial}(e, x)_\pi$.

Hence, the algorithm given finds

$$\bigcup_{x \in \mathrm{params}(f)} \left(\bigcup_{\pi \in \Pi} \mathrm{initial}(e, x)_\pi \right).$$

Potential Variables continued

The set $\bigcup_{x \in \mathsf{params}(f)} \left(\bigcup_{\pi \in \Pi} \mathsf{initial}(e, x)_\pi \right)$ does not contain information about which paths some edges form initial assignments on and, in fact, keeping track of this would be costly.

Instead, we acknowledge that $\bigcup_{x \in \mathsf{params}(f)} \left(\bigcup_{\pi \in \Pi} \mathsf{initial}(e, x)_\pi \right)$ is a set of potential initial assignments.

For some e that performs an initial assignment on a path π, there may be another path going through e on which e is not the first assignment.

We address this in instrumentation simply by asking if a value has already been recorded.

Cases not yet dealt with

```
a = A()
b = a; b.attr = "some␣value"
r = f(a.attr)
```

f depends on *a*, but *b* holds the same reference, hence the assignment to `b.attr` is the same as an assignment to `a.attr`.

Some effort should be made to deal with reference tracking (though I don't expect to do it perfectly since we want to minimise overhead).

VyPR doesn't do any reference tracking *yet* (since this 1) could seriously damage the performance of the monitored program and 2) may not agree with CFTL's motivation), so a limited amount here is fine.

Cases not yet dealt with

```
a = args[0]
r = a
for i in range(10):
  r = r + f(i)
g(r)
```

```
a = args[0]
r = a
for i in range(10):
  r = r + f(i)
  g(r)
```

For the left example, the initial value of *i* is the first value of the loop, but for the right example, it's the value in the current iteration.

Possible criteria: if the first statement to use an inherited variable is reachable from the function call we care about (ie, by following a loop around), we take the value of the current iteration. If not, we take the value from the first iteration.

Cases not yet dealt with

```
1  a = args[0]
2  r = a
3  for i in range(10):
4    if P(i):
5      i = i + 1
6    r = r + f(i)
7  g(r)
```

On line 6, is *i* inherited or not? It depends on the predicate *P*, which in generality cannot be assumed to be statically computable.

Our existing notion of inherited variables is not enough since it implies certainty; in this case, the inherited status is branch dependent.

Possible solution: Replace our definition of inherited with potentially inherited and add markers to assignment statements to flag that an inherited variable has ceased to be inherited. This way, we gain information from the current run.

The First Implementation

Output from VyPR after monitoring a program (details of which aren't that helpful):

```
VERDICT REPORT with respect to
Forall(OrderedDict([('q', q = StaticState(changes=a))])).\
Formula(d(next_transition(q = StaticState(changes=a), h)) in [0, 1])
--------------------------------------------------
Binding
[
state change resulting from assignment to a : line 13
]
gave 2 verdicts, 2 true and 0 false:

        True at 2019-01-24 12:35:39.802515 with observations
                {
                d(next_transition(q = StaticState(changes=a), h)) in [0, 1]:
                datetime.timedelta(0, 0, 66)
                }
        and affecting variable values
                {d(next_transition(q = StaticState(changes=a), h)) in [0, 1]:
                        {'a': 2, 'o': 3, 'n': 0, 'r': 4, 'u': 3, 'v': 2}}

        True at 2019-01-24 12:35:39.802651 with observations
                {
                d(next_transition(q = StaticState(changes=a), h)) in [0, 1]:
                datetime.timedelta(0, 0, 6)
                }
        and affecting variable values
                {d(next_transition(q = StaticState(changes=a), h)) in [0, 1]:
                        {'a': 2, 'o': 3, 'n': 1, 'r': 4, 'u': 3, 'v': 2}}
```

Next Steps - Analysis

With this work, we have a mapping from points in code ($\langle \beta, \alpha \rangle$ tuples) to observation severity and initial variable values.

We can go in both directions:

For a fixed valuation of some variables, which points in code that we monitor have these as initial variables, and therefore what do we observe from those points wrt our CFTL specification?
eg, for given variable values, what does the severity of failure look like for a given function call?

For a fixed point in code, what are the initial variables, and what is the relationship between their values and the severity of failure generated by this point in code?

Next Steps - Analysis continued

Collecting this data over time, we can perform detailed analysis on the relationship between state and time in a program.

Possibly make predictions...

Next Steps - Optimisation

Storing this amount of state data for monitoring requires a lot of space overhead.

It is vital that we optimise storage.

The first place to look is reducing overlap, for example, it is likely that function calls will share initial variables... there is no need to store the values separately for each event that uses them.

Next Case Studies

Have begun to discuss testing VyPR with the pdmv team.

Potentially the Conditions Browser, as well.

Overall, we will have good coverage of the Python side of the "physics ecosystem".