We present Execution Indexing as a formal approach to unique identities within and between program executions and provide one intuitive approach to representing identity that has proven useful in improving the correctness or precision of real world analyses.

Structural Execution Indexing is a specific, novel approach to correlation within a program’s execution. Structural Execution Indexing uses fundamental to dynamic program analyses is the notion of identity for a point within an execution. A lack of formalization has led to improving the correctness or precision of real world analyses.

One Program

Two Iterations Zero Iterations

How Do We Know These Points Correspond?

Recognizing the same point between both executions can be useful in various analyses, but the meaning of “same” depends on context.

Structural Execution Indexing uses dynamic control dependence

What If Data Determines Identity Semantics?

e.g. event based actions (on input ‘a’ do...)

Semantic Anchor Points allow the derivation tree to be re-rooted at a value for the duration of control flow centered upon that value.

1) Context

Fundamental to dynamic program analyses is the notion of identity for a point within an execution. A lack of formalization has led to imprecise and heuristic approaches for representing execution points. This makes analyses inherently imprecise and less useful.

We present Execution Indexing as a formal approach to unique identities within and between program executions and provide one intuitive approach to representing identity that has proven useful in improving the correctness or precision of real world analyses.

2) Introduction to Structural Execution Indexing

Execution Indexing is a general theory of uniquely identifying points of correspondence between program executions or points of correlation within a program’s execution.

Structural Execution Indexing is a specific, novel approach to Execution Indexing utilizing equivalence and similarity between dynamic control dependences of any points in an execution.

One Program

Two Iterations Zero Iterations

How Do We Know These Points Correspond?

Recognizing the same point between both executions can be useful in various analyses, but the meaning of “same” depends on context.

Structural Execution Indexing uses dynamic control dependence

What If Data Determines Identity Semantics?

e.g. event based actions (on input ‘a’ do...)

Semantic Anchor Points allow the derivation tree to be re-rooted at a value for the duration of control flow centered upon that value.

3) Formalizing Structural Indices

An Execution Description Language is a context-free grammar that expresses all possible executions of a program. e.g.

<table>
<thead>
<tr>
<th>Construct</th>
<th>EDL Representation</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) s1;</td>
<td>S → ε</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>2) s1;</td>
<td>S → 1R1</td>
<td>1 2 4</td>
</tr>
<tr>
<td>3) if (C1) {</td>
<td>S → 1R1</td>
<td>1 2 4</td>
</tr>
<tr>
<td>4) s2;</td>
<td>S → 2R8</td>
<td>2 5</td>
</tr>
</tbody>
</table>

Identifying EDL Terminals is Equivalent

Consider the corresponding executions of statement 10 in the two executions below:

4) Precise Debugging

Debugging often utilizes breakpoints to freeze an execution at certain specific times for analysis. The more precise these breakpoints are, the more precise and the faster the debugging takes place.

Particularly with the growth of automated debugging, where breakpoints may be set and analyzed by computers themselves, consistency and correctness are crucial.

5) Dynamic Extraction of Control Structures

Several modern analyses rely on dynamically extracting control structures to refine future actions or winnow aggregated information.

When does the iteration we just started end?

This forms a crucial part of:
- Hierarchical Dynamic Slicing
- Execution Omission Error Detection
- Potential Concurrency Profiling

6) Data Race Classification

A data race: 2 Memory accesses, at least 1 write
- Multiple threads
- No ordering constraints

They can cause nondeterministic execution.

Existing detection methods warn unnecessarily.

Classify to prioritize developer efforts:
- Re-execute and force race expression to classify:

Selecting Dynamic Accesses for Testing

Numerous dynamic memory accesses exhibit the same semantic race: Infeasible to test them all
- Categorize by structure and data semantics
- Force exhibition of maximally distant candidates

References