Abstract Semantic Differencing via Speculative Correlation

NIMROD PARTUSH
ERAN YAHAV
TECHNION (HAIFA, ISRAEL)
[OOPSLA’14]
Problem: Semantic Differencing

For two procedures P, P’:

- Prove that P and P’ are equivalent
  - Produce the *same output for the same input*

- Otherwise, produce a *useful* description of the difference
Motivation

- “Successful software always gets changed.”
  – Frederick P. Brooks

- Program understanding & Debugging

- Test generation & pruning

- “Big Code”
Example: Sequence Printing

Prints a sequence of numbers from \textit{first} to \textit{last} in increments of \textit{STEP}

- Simplified code from \texttt{Coreutils seq.c} version 6.9
- For instance for the input \textit{first} = 1, \textit{last} = 13, \textit{STEP} = 2 the procedure will print the sequence 1, 3, 5, 7, 9, 11, 13

```c
static void
print_numbers_v6_9(long first, long last, ...)
{
    long i;
    for (i = 0; /* empty */; i++) {
        long x = first + i * STEP;
        if (last < x) break;
        if (i) fputs (separator, stdout);
        printf (fmt, x);
    }
    if (i)
        fputs (terminator, stdout);
}
```
Example: Sequence Printing

Does version 6.10 print the same sequence?

Syntactic difference is vast

But the output is the same: x is equivalent at the point of printing (9,10')
Test Equivalence

- Run the procedures with the same inputs and check outputs for equivalence

$\text{first} = 1, \text{last} = 3, \text{STEP} = 1$

- Can only show cases where procedures differ
  - Modulo cost and coverage (if you are lucky)
  - Cannot prove equivalence for most programs
Abstract Semantic Differencing

- Use **abstract interpretation** to **prove equivalence** between two program versions
- Or **characterize their difference**
  - Find (an abstraction of) **differing programs states** that come from the same input

**Sound**
- Equivalence is guaranteed

**Precise**
- Report few false differences
Equivalence Under Abstraction

- Analyzing the programs separately may result in false equivalence
  - For instance, interpreting with a numerical relational abstraction:

\[
\{ i \leq n \} \neq \{ i' \leq n' \}
\]

Equivalence under abstraction does not entail equivalence between the concrete values it represents.
Our Approach

- Analyze $P$ and $P'$ **together**
  - Define a *correlating semantics* that interprets the programs together
  - Interpretation is done in some *interleaving* of their steps
  - **Abstract** the correlating semantics to handle infinite-state programs

- **Search** for the interleaving that allows the abstraction to best capture equivalence
Correlating Semantics

- Maintain **direct correlation** between values in the programs P and P’
  - Use a **relational** abstraction that **captures equivalences**

\[ \sigma = \sigma \{ i \leq \{i\} \neq i' \leq \sigma h = \{i''\} \leq n' \} \]
Order Matters!

- Analysis order determines the abstract correlating semantics’ ability to track equivalence
  - For example, in **sequential** order:
    - By the time one program’s analysis is finished, the values have been abstracted and equivalence is lost
Choosing Program Interleaving

- In which order should the programs be analyzed?
- Sequential?

![Diagram showing program execution with abstract states and interpretation steps.](attachment:image.png)

```plaintext
while (i < n) {
    i += 1;
    print(i);
}
```

ACS 12
Choosing Program Interleaving

- In which order should the programs be analyzed?
  - Lock-Step?

Abstract state

Interpretation step

```c
static void
print_numbers_v6_10(long first, long last, ...)
{
  bool out_of_range = (last < first);
  if (!out_of_range) {
    long x = first;
    long i;
    for (i = 1; /* empty */; i++) {
      long x0 = x;
      printf (fmt, x);
      x = first + i * STEP;
      out_of_range = (last < x);
      if (out_of_range) break;
      fputs (separator, stdout);
    }
    fputs (terminator, stdout);
  }
}

print(i):
static void
print_numbers_v6_9(long first, long last, ...)
{
  long i;
  for (i = 0; /* empty */; i++) {
    long x = first + i * STEP;
    if (last < x) break;
    if (i) fputs (separator, stdout);
    printf (fmt, x);
  }
  if (i)
    fputs (terminator, stdout);
```
Choosing Program Interleaving

- In which order should the programs be analyzed?
  - All possible interleavings?
    - Will result in an exponential blow-up

Abstract state interpretation step
Choosing Program Interleaving

- The challenge is to find an interleaving that allows maintaining equivalence under abstraction
- While avoiding exponential blow-up
Speculative Exploration

- The search for an interleaving is part of the fixed-point abstract interpretation analysis.
- The search drives the analysis.
- The algorithm is composed of two steps.

```
while (worklist ≠ ∅)
    results ← Speculate(P, P', worklist, state北大, k)
    (worklist, state北大) ← Find_Minimal_Diff_Result(P, P', results)
```

- Speculatively explore interpreting k steps, distributed over both programs.
  - 0 over P & k over P'
  - 1 over P & k − 1 over P'
  - Etc.
- k is the speculation window.
- A parameter to the algorithm.

At the end of each speculative step, compare the k+1 results and pick the one with minimal difference.
Speculative Exploration: Example

Speculate \((k = 7)\)

```c
static void print_numbers_v6_9(long first, long last, ...)
{
    long i;
    for (i = 0; /* empty */; i++) {
        long x = first + i * STEP;
        if (last < x) break;
        if (i) fputs (separator, stdout);
        printf (fmt, x);
    }
    if (i)
        fputs (terminator, stdout);
}
```

0 steps over v6.9,
7 steps over v6.10

\((4,11')\) ⇒
\[
\{i' = 1, i = ?, \quad \text{x' = first' + STEP',} \quad \text{x = ?,} \quad ...
\}
```

3 steps over v6.9,
4 steps over v6.10

\((7,7')\) ⇒
\[
\{i' = 1, i = 0,
\text{x' = x = first,} \\
\text{...}
\}
```

7 steps over v6.9,
0 steps over v6.10

\((6,4')\) ⇒
\[
\{i' = ?, i = 1, \\
\text{x' =?,} \quad x = \text{first,} \\
\text{...}
\}
Comparing Abstractions to find Minimal Difference

We explored two strategies

1. Scale-oriented: count the number of equivalent variables (denoted $\equiv_{\{v\}}$) per state

2. Precision-oriented: Use the abstract domain (APRON Polyhedra) geometrical representation to compute difference inclusion

\[
\sigma_{\bowtie 1} = \\
(9,4') \mapsto \{ i = 0, i' = ?, x = \text{first}, x' = ?, x \leq \text{last}, x'_{0} = x' \} \\
(6,7') \mapsto \{ 0 \leq i, i' = 1, \text{first} \leq x \leq \text{last}, x' = \text{first} \} \\
\ldots 
\]

\[
\sigma_{\bowtie 2} = \\
(5,5') \mapsto \{ i = i' = 1, x = ?, x' = \text{first} + \text{STEP}, x'_{0} = ? \} \\
(7,9') \mapsto \{ 0 \leq i = i', x \leq \text{last}, x' = x'_{0} \} \\
\ldots 
\]
Speculative Exploration: Visualization ($k = 7$)
The speculative algorithm is geared towards finding minimal difference (and not \textit{strictly} equivalence).

A usable description of difference is produced even if full equivalence does not hold.

\[(9, 10') \mapsto \begin{cases} 
\{\text{print\_extra\_number}, x' = x, i' = i + 1, x \leq \text{last}\} \\
\lor \\
\{\neg\text{print\_extra\_number}, x' = x + \text{STEP}, i' = i + 1, x \leq \text{last}\} 
\end{cases}\]
## Evaluation

<table>
<thead>
<tr>
<th>Function</th>
<th>#LOC</th>
<th>#Patch</th>
<th>#Loops</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_numbers</td>
<td>23</td>
<td>7-13+</td>
<td>1</td>
<td>00:11 (k=2)</td>
</tr>
<tr>
<td>cache_fstatat</td>
<td>17</td>
<td>2-4+</td>
<td>0</td>
<td>00:03 (k=1)</td>
</tr>
<tr>
<td>set_owner</td>
<td>51</td>
<td>2-4+</td>
<td>0</td>
<td>00:02 (k=2)</td>
</tr>
<tr>
<td>fmt</td>
<td>42</td>
<td>5-5+</td>
<td>1</td>
<td>00:22 (k=2)</td>
</tr>
<tr>
<td>md5sum</td>
<td>40</td>
<td>0-3+</td>
<td>3</td>
<td>13:31 (k=2)</td>
</tr>
<tr>
<td>char_to_clump</td>
<td>111</td>
<td>2-12+</td>
<td>3</td>
<td>19:09 (k=2)</td>
</tr>
<tr>
<td>savewd</td>
<td>86</td>
<td>0-1+</td>
<td>0</td>
<td>00:46 (k=2)</td>
</tr>
<tr>
<td>addr</td>
<td>77</td>
<td>1-2+</td>
<td>0</td>
<td>00:17 (k=1)</td>
</tr>
<tr>
<td>SetTextInternal</td>
<td>47</td>
<td>0-3+</td>
<td>1</td>
<td>11:28 (k=3)</td>
</tr>
<tr>
<td>get_shal_basic v1</td>
<td>145</td>
<td>3-10+</td>
<td>2</td>
<td>118:01 (k=2)</td>
</tr>
<tr>
<td>get_shal_basic v2</td>
<td>149</td>
<td>2-20+</td>
<td>2</td>
<td>TO (2H)</td>
</tr>
<tr>
<td>get_path_prefix</td>
<td>22</td>
<td>2-3+</td>
<td>1</td>
<td>29:12 (k=3)</td>
</tr>
<tr>
<td>boot_attr v1</td>
<td>77</td>
<td>7-4+</td>
<td>0</td>
<td>08:08 (k=4)</td>
</tr>
<tr>
<td>boot_attr v2</td>
<td>74</td>
<td>5-7+</td>
<td>0</td>
<td>06:04 (k=4)</td>
</tr>
<tr>
<td>read_attr</td>
<td>32</td>
<td>1-4+</td>
<td>1</td>
<td>05:42 (k=2)</td>
</tr>
<tr>
<td>ll_binary_merge</td>
<td>37</td>
<td>8-24+</td>
<td>1</td>
<td>00:53 (k=1)</td>
</tr>
<tr>
<td>write_zip_entry</td>
<td>340</td>
<td>1-4+</td>
<td>3</td>
<td>07:32 (k=2)</td>
</tr>
<tr>
<td>DDEC</td>
<td>10</td>
<td>3-3+</td>
<td>1</td>
<td>00:13 (k=1)</td>
</tr>
<tr>
<td>DSE</td>
<td>7</td>
<td>2-3+</td>
<td>1</td>
<td>00:09 (k=1)</td>
</tr>
<tr>
<td>RegVer</td>
<td>10</td>
<td>4-4+</td>
<td>1</td>
<td>00:07 (k=1)</td>
</tr>
<tr>
<td>SymDiff</td>
<td>32</td>
<td>5-4+</td>
<td>0</td>
<td>00:04 (k=1)</td>
</tr>
</tbody>
</table>
The interleaving/matching problem is closely coupled with semantic diff & equivalence

- Previous approach mainly use syntactic cues

New approach: search for an interleaving

- With an equivalence criteria to drive the search

A description of difference is produced, instead of a binary yes/no for equivalence

- Useful for program understanding, debugging etc.

Available on github

Contact us!
{nimi,yahave}@cs.technion.ac.il
Proposed Questions

1. Shouldn’t there (still) be some exponential blow-up here??
2. Where does the variable matching come from?
3. Can you talk about how related work find their interleaving?
4. How do you handle function calls?
5. How do you handle heap array data?
6. Why were the $k$’s in your evaluation so small?
7. Can you elaborate on the precise method for comparing abstractions?
8. You seem to have used a disjunctive domain, how did you $\sqcup$ and $\sqcap$ it?
9. Won’t you miss interleavings?
10. How does the approach scale (inter-procedurally)?
11. Is the analysis proportional to the size of change program both?
Un-interpreted Functions

- Function calls are handled modularly
  - If $foo$ was proven to be equivalent, and so are $y$ & $z$, the result will be equivalent
    - Otherwise, anything (T) is possible for $x$
      \[
      \equiv foo \equiv \{v_1, v_2\} \\
      \equiv foo(v_1, v_2)
      \]

- Array and heap access are modeled similarly
  \[
  \equiv array \equiv idx \\
  \equiv array[idx]
  \]
Related work: program matching

- Symbolic Execution based approaches [DSE, UC-KLEE]:
  - Bound loop iteration
  - try to explore all bound paths
  - Check equivalence on each path

- Recursion-Rule based approaches [SymDiff]:
  - Transform loops to function calls
  - Use function calls as matching locations
    - i.e. try and prove the inputs to each function call are equivalent

- Data\Trace based approaches [DDEC]:
  - Use run traces variable values to infer a bi-simulation relation of the two programs
Equivalence-based Partitioning

- We use a disjunctive domain to allow separating **equivalent paths** from **differing paths**
  \[ \{ \text{print\_extra\_number}, x' = x, i' = i + 1, x \leq \text{last} \} \]
  \[ \lor \]
  \[ \{ \neg \text{print\_extra\_number}, x' = x + \text{STEP}, i' = i + 1, x \leq \text{last} \} \]

- How do you Join such a domain?
  - which sub-states are joined with which?

- We Join and Widen abstract states based on the **equivalences** they **preserve**
  - the set of variables that hold equivalence
  - disjunction size bound at \(2|\text{VAR}|\)
  - lose some information, but maintain what's important (equivalence)
The abstract differences $\Delta_1, \Delta_2$ is computed for both states, and then containment $\Delta_1 \sqsubseteq \Delta_2$ (or vice-versa) is checked.
Partial Order Reduction

- Exploring **all interleavings** over two programs within $k$ steps would yield $2^k$ results
  - Since each step can be performed on either $P$ or $P'$
  - For the most part, a single representative is sufficient
    - Meaning foreach $0 < i \leq k$ we explore first interpreting all $i$ steps over $P$ and then $k - i$ over $P'$
      - No alternation in-between
      - Resulting in $k + 1$ results
    - Since the domain is commutative and join\widen is only performed after each speculative step
  - This does mean we miss some interleavings
    - But did not pose an issue in our benchmarks
    - A small sacrifice to make for scaling
For full program equivalence, we currently use the modular approach

- And in the case the callee differs among versions, we assume T
  - Future: use the description of difference instead, somehow

This is the state of the art for equivalence checking

- The only full-program approaches (known to this presenter :) are symbolic execution based ones, that try to pin-point differing paths
  - Usually get poor coverage
  - Generally unable to prove equivalence
int foo(int x, int y) {
    int z = 0, w, o;
    z++;
    ...
    for (x = 0; x < 2*y; ++x)
        w += 2;
    ...
    if (y > 42) {
        z -= 13;
        o = z + zoo(3);
        x--;
    }
    ...
}

int foo`(int x`, int y`) {
    int z` = 1, w`, o`;
    z`--;
    ...
    for (x` = 0; x` < y`; ++x`)  
        w` += 4;
    ...
    if (y` > 47) {
        z` -= 12;
        o` = z` + zoo(3);
        x`++;  
    }
    ...
}