Fidgeting Till The Point Of No Return

Marina Biberstein
Eitan Farchi
Shmuel Ur
Table of contents

Background: problems and existing solutions

Fidgeting: why and how

Summary
A sample program

\[
x = 1 \\
y = 1
\]

Many possible interleavings
Only few are generated by the test environment

\[
t_1 = x \\
t_2 = y
\]

if \( t_1 \neq 1 \) goto L1

print \( t_1 \)

L1: print \( t_2 \)

\[
x = 3 \\
y = 3
\]
Making things happen – the noise-making tools

```plaintext
x=1
y=1

t1=x
t2=y

if (t1!=1) goto L1
print t1

L1: print t2

x=3
y=3

if (random()>P) yield();
```
Making things happen – the noise-making tools

if (random()>P)
    yield();

if (t1!=1) goto L1
L1: print t2
Making things happen – the noise-making tools

x=1
y=1
t1=x
t2=y
if (t1!=1) goto L1
print t1
if (random()>P) yield();

Difficult to change order of distant events
Many changes don’t affect the outcome

L1: print t2
x=3
y=3
Noise-making tools: equivalent schedules

```
if (random()>P) yield();

x=1
y=1
t1=x
t2=y
if (t1!=1) goto L1
print t1
L1: print t2
x=3
y=3
```
Noise-making tools: equivalent schedules

if (random()>P) yield();

x=1
y=1
t1=x
x=3
t2=y
if (t1!=1) goto L1
print t1
L1: print t2
y=3
Noise-making tools: equivalent schedules

if (random()>P) yield();

x=1
y=1
t1=x
t2=y

if (t1!=1) goto L1

print t1

x=3
y=3

L1: print t2
if (random()>P) yield();

x=1
y=1
t1=x
t2=y
if (t1!=1) goto L1
print t1
L1: print t2
x=3
y=3
Alternative Pasts: generating interesting things

```
x=1
y=1

if (t1!=1) goto L1
print t1

L1: print t2

x=3
y=3
x=0
y=0
```

Initialization

Present

Future
Alternative Pasts: generating interesting things

$t1=x$

$x=1$
$y=1$

$t2=y$

if ($t1 
eq 1$) goto L1

print $t1$

x=3
y=3

L1: print $t2$
Alternative Pasts: generating interesting things

Issues

- How to delay events selection?
- What is the smart choice of values?

Advantages

- Generates interleavings that are significantly different
- Easier to swap distant events

```
x=1
y=1
t1=x
t2=y
if (t1!=1) goto L1
print t1
L1: print t2
x=3
y=3
```
Table of contents

Background: problems and existing solutions

Fidgeting: why and how

Summary
Looking for solutions

Advantages:

- More time for values to arrive
- Better understanding of what values are interesting

Intuition

- Move value selection to a “decision point”

```plaintext
x = 1
y = 1

x = 0
y = 0

x = 3
y = 3

if (t1 != 1) goto L1
print t1
L1: print t2
```
Fidgeting: the basic concepts

- Instructions: broken into two groups
  - Can be re-executed: =, +, -, ...
  - Can’t be re-executed: if, print

- Events:
  - Instruction
  - Variables read
  - Variable written

- Visibility graph:
  - Timing restrictions on events
  - Nodes:
    - Event
    - Event state (raw or processed)
  - Edges: timing precedence

\[ t = \text{read}(); \]
\[ t++; \]
Visibility: When can a value be used?

Problem:
- Node $r$ reads variable $\lambda$
- Node $w$ writes variable $\lambda$
- Can $r$ use the value produced by $w$?

Answer: Yes, unless timing restrictions in visibility graph imply that
- $r$ precedes $w$, or
- Another node that writes $\lambda$ intervenes between $w$ and $r$

In graph terms:
- There is a path from $r$ to $w$, or
- There is a path from $w$ to $r$ that passes through a node writing $\lambda$
Hiding nodes

- **Situation:**
  - Node \( r \) reads variable \( \lambda \)
  - Nodes \( w, w' \) write variable \( \lambda \) and are visible from \( r \)

- **Problem:** make \( w' \) invisible

- **Solution:**
  - Add edge \((r, w')\), or
  - Add edges \((w', w)\) and \((w, r)\)

- Exists a method that doesn’t introduce cycles
Processing node

- Goal: Select the values to be used by node $n$
- Processing node $n$:
  - If node state is *processed* – done
  - Set node state to *processed*
  - For every variable $\lambda$ read by $n$
    - Select a visible node $w$ that writes $\lambda$
    - Hide all other visible nodes that write $\lambda$
    - Process $w$
Fidgeting: An outline

- Start executing the tested program
- At each event:
  - Create a new raw node
  - Add it to graph
    - First event in thread:
      - Add edge from create in the parent thread
      - Add edges from initialization events
    - Otherwise: add edge from the previous event in the thread
  - If the instruction cannot be replayed: process the node
  - Execute the event,
    - Raw: no intervention
    - Processed: for each read variable, use its value as produced by the visible write event
Fidgeting around

```plaintext
x = 1
y = 1

x = 3
y = 3

t1 = x

if (t1 != 1) goto L1

print t1

L1: print t2

x = 3
y = 3
```

Diagram:

- Start at (x=1, y=1)
- Move to (x=0, y=0)
- Go to t1 = x
- Check if t1 != 1, if true go to L1
- Print t1
- Go to L1: Print t2
- End at (x=3, y=3)
Fidgeting around

```
x=1
y=1

x=1
y=1
```

```
x=3
y=3
```

```
t1=x
```

```
x=3
y=3
```

```
t2=y
```

```
if (t1!=1) goto L1
```

```
print t1
```

```
L1: print t2
```

```
x=3
```

```
y=3
```

```
t2=y
```

```
if (t1!=1) goto L1
```

```
print t1
```

```
L1: print t2
```

```
x=3
```

```
y=3
```

```
t2=y
```

```
if (t1!=1) goto L1
```

```
print t1
```

```
L1: print t2
```

```
x=3
```

```
y=3
```
Table of contents

Background: problems and existing solutions

Fidgeting: why and how

Summary
Summing up

- A new algorithm for generating interesting interleavings
- More aggressive delays that with alternative pasts
- More informed choice of values at decision points
  - Especially useful for achieving coverage
- Noise-makers can help delay decision points
- Complexity issues remain to be addressed
  - Some optimizations available and should be evaluated
There once was a man who said, “God must think it exceedingly odd if He finds that this tree continues to be when there’s no one about in the Quad.”

“Dear Sir:
Your astonishment’s odd:
I am always about in the Quad
And that’s why the tree
Will continue to be,
Since observed by,
Yours faithfully,
God.”