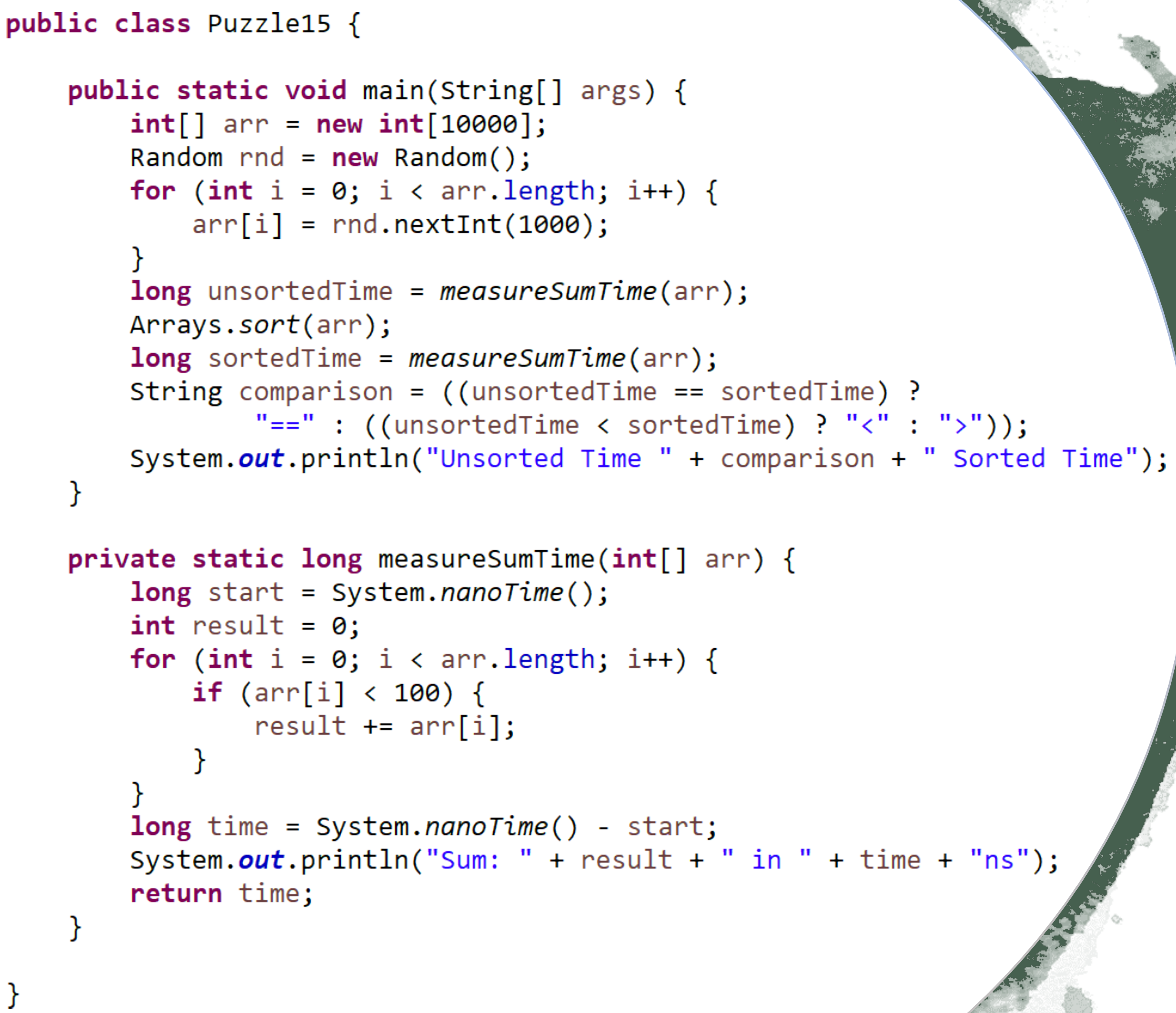
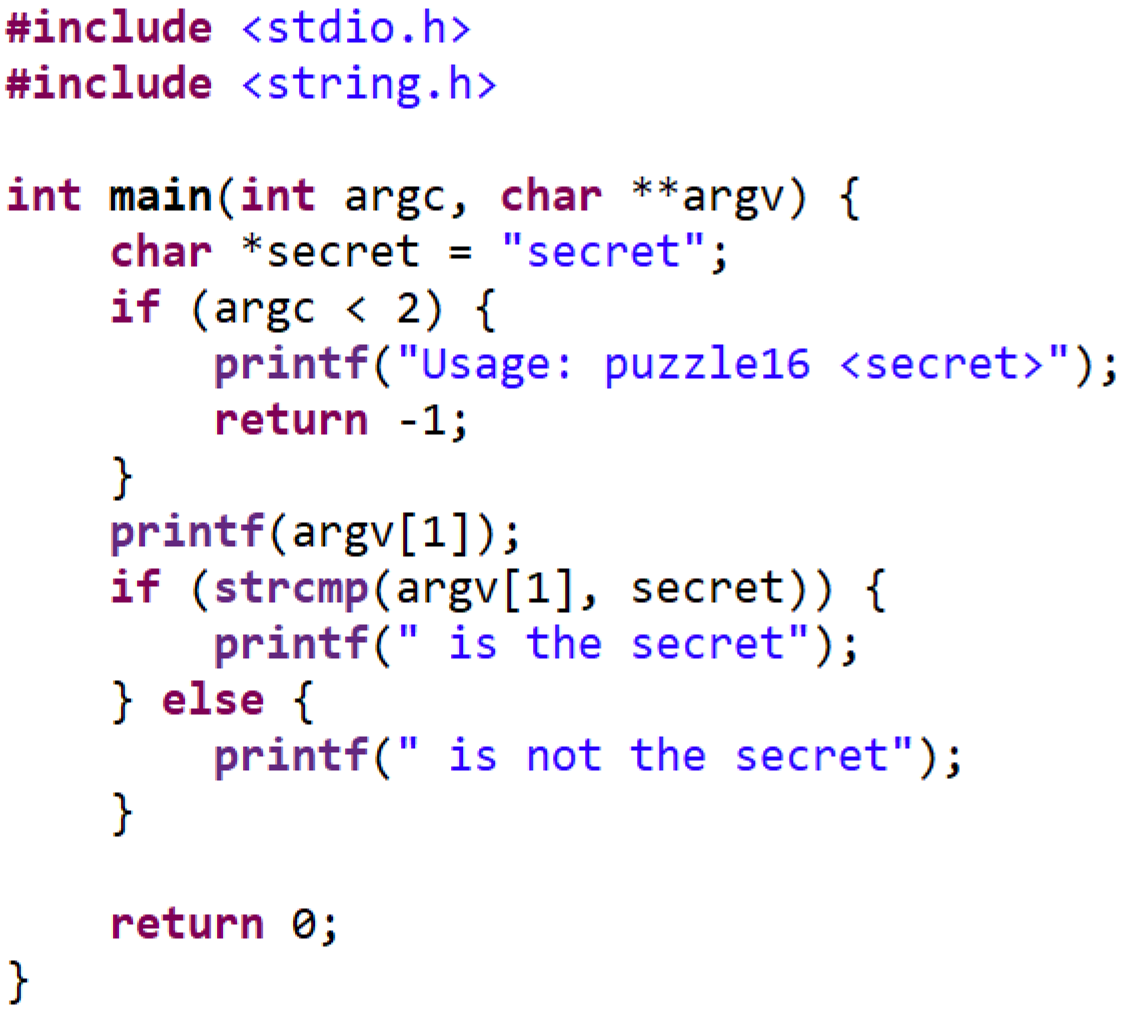


```java
public class Puzzle15 {

    public static void main(String[] args) {
        int[] arr = new int[10000];
        Random rnd = new Random();
        for (int i = 0; i < arr.length; i++) {
            arr[i] = rnd.nextInt(1000);
        }
        long unsortedTime = measureSumTime(arr);
        Arrays.sort(arr);
        long sortedTime = measureSumTime(arr);
        String comparison = ((unsortedTime == sortedTime) ?
                "==" : ((unsortedTime < sortedTime) ? "<" : ">"));
        System.out.println("Unsorted Time " + comparison + " Sorted Time");
    }

    private static long measureSumTime(int[] arr) {
        long start = System.nanoTime();
        int result = 0;
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] < 100) {
                result += arr[i];
            }
        }
        long time = System.nanoTime() - start;
        System.out.println("Sum: " + result + " in " + time + "ns");
        return time;
    }

}
```



```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char *secret = "secret";
    if (argc < 2) {
        printf("Usage: puzzle16 <secret>");
        return -1;
    }
    printf(argv[1]);
    if (strcmp(argv[1], secret)) {
        printf(" is the secret");
    } else {
        printf(" is not the secret");
    }

    return 0;
}
```

# Background

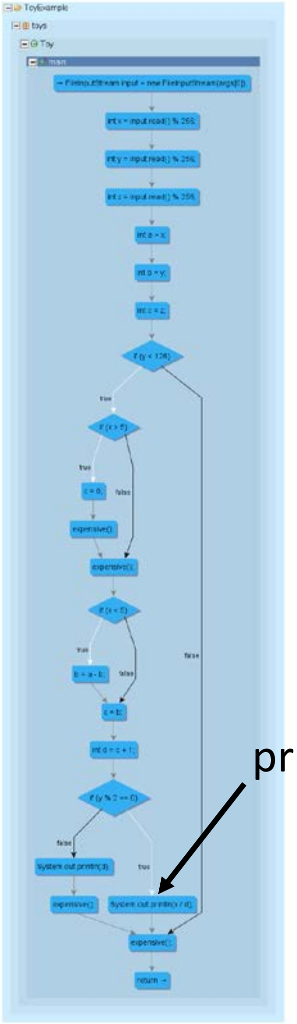
- We research tools and techniques that:
  - Produce human-verifiable and comprehensible evidence
  - Are scalable
- Resulted in key concept: Projected Control Graph (PCG)
  - Retains minimal necessary program behaviors
- Verification of lock/unlock pairs in Linux
  - Over 66,000 instances (3 versions of Linux)
  - PCG based tool has resulted in 8 bug reports which have been fixed

# Motivation behind PCGs

- Control Flow Graphs can be HUGE
  - CFG of a single function in Linux
    - lustre_assert_wire_constants function has 2^656 paths!
  - Programmers don’t play dice…
  - Is all of this complexity necessary to solve a given analysis problem?
  - Is there an underlying design pattern that can be leveraged to complete the analysis?

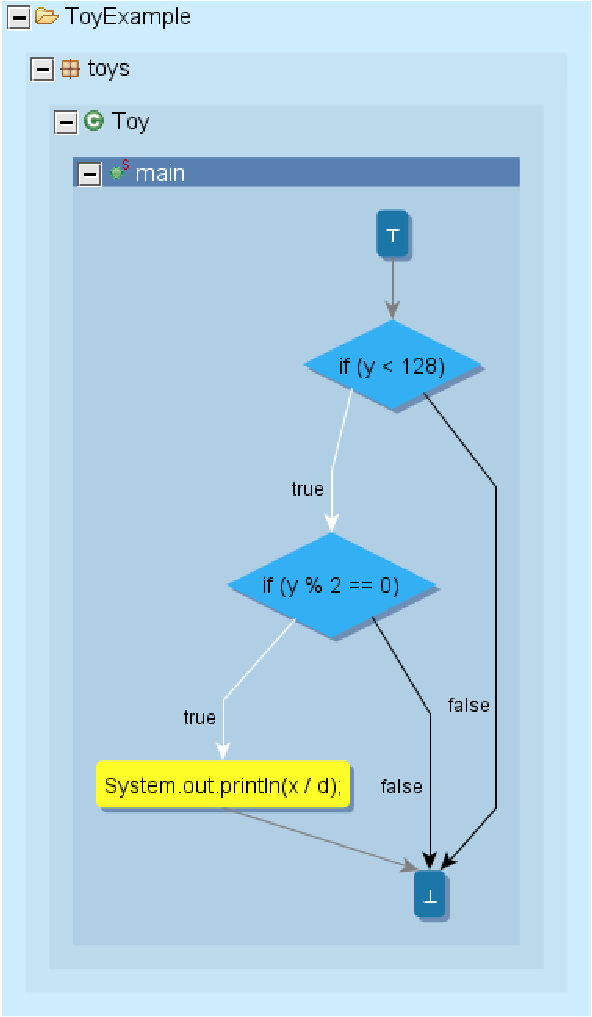
# Concept: Projected Control Graph (PCG)

- Proposed by Ahmed Tamrawi in 2016
- Efficiently groups program behaviors into equivalence classes of homomorphic behaviors
    - Parameterized by events of interest
    - Only event statements and necessary conditions are retained
- Result is a compact structure-preserving model of a CFG w.r.t. events of interest



print(x / d)

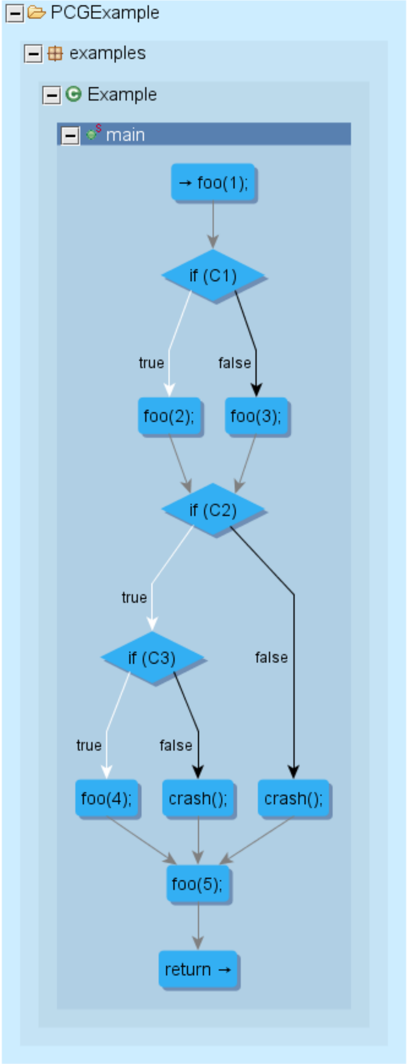
CFG



PCG(E), E=Division Statement

# PCG Example

```
public void main() {
    foo(1);
    if (C1) {
        foo(2);
    } else {
        foo(3);
    }
    if (C2) {
        if (C3) {
            foo(4);
        } else {
            crash();
        }
    } else {
        crash();
    }
    foo(5);
    return;
}
```

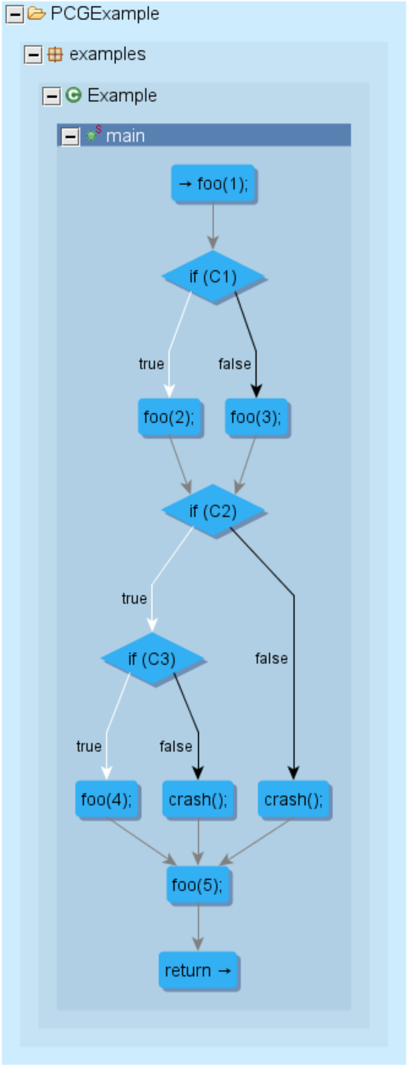


$2^3$=8 possible values for the tuple (C1, C2, C3)

| C1 | C2 | C3 | Behavior |
|---|---|---|---|
| False | False | False | |
| False | False | True | |
| False | True | False | |
| False | True | True | |
| True | False | False | |
| True | False | True | |
| True | True | False | |
| True | True | True | |

# PCG Example

```
public void main() {
    foo(1);
    if (C1) {
        foo(2);
    } else {
        foo(3);
    }
    if (C2) {
        if (C3) {
            foo(4);
        } else {
            crash();
        }
    } else {
        crash();
    }
    foo(5);
    return;
}
```

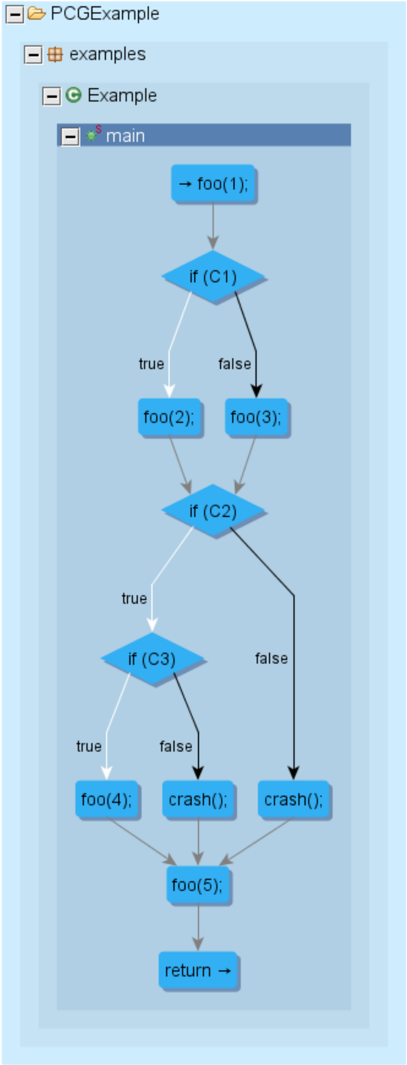


If C2 is false then C3 is not evaluated.

| C1 | C2 | C3 | Behavior |
|---|---|---|---|
| False | False | N/A | |
| False | False | N/A | |
| False | True | False | |
| False | True | True | |
| True | False | N/A | |
| True | False | N/A | |
| True | True | False | |
| True | True | True | |

# PCG Example

```
public void main() {
    foo(1);
    if (C1) {
        foo(2);
    } else {
        foo(3);
    }
    if (C2) {
        if (C3) {
            foo(4);
        } else {
            crash();
        }
    } else {
        crash();
    }
    foo(5);
    return;
}
```

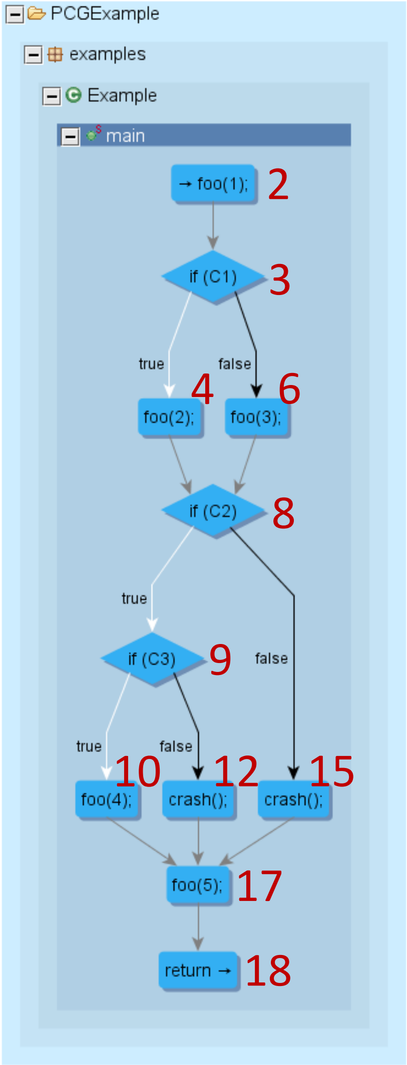


CFG has 6 paths.

| C1 | C2 | C3 | Behavior |
|---|---|---|---|
| False | False | N/A | |
| False | True | False | |
| False | True | True | |
| True | False | N/A | |
| True | True | False | |
| True | True | True | |

# PCG Example

```
 1  public void main() {
 2      foo(1);
 3      if (C1) {
 4          foo(2);
 5      } else {
 6          foo(3);
 7      }
 8      if (C2) {
 9          if (C3) {
10              foo(4);
11          } else {
12              crash();
13          }
14      } else {
15          crash();
16      }
17      foo(5);
18      return;
19  }
```

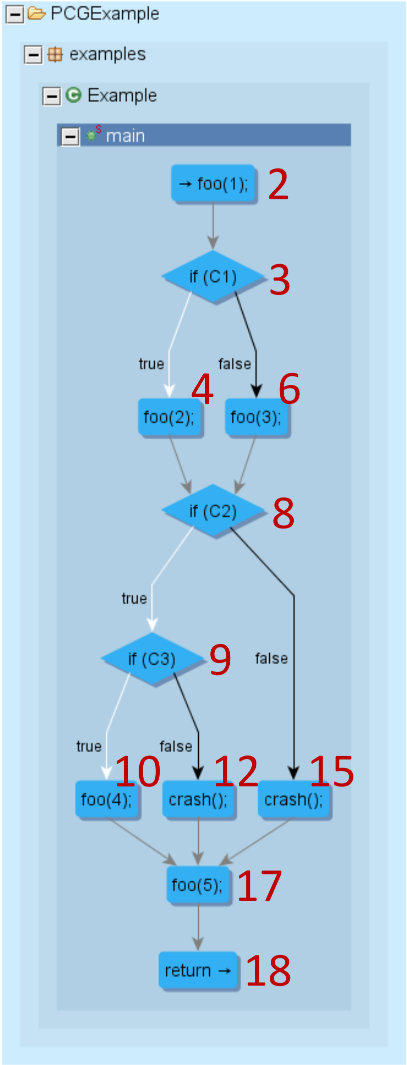


Which paths include a “crash” event?

| C1 | C2 | C3 | Behavior |
|---|---|---|---|
| False | False | N/A | 2,3,6,8,15,17,18 |
| False | True | False | 2,3,6,8,9,12,17,18 |
| False | True | True | 2,3,6,8,9,10,17,18 |
| True | False | N/A | 2,3,4,8,15,17,18 |
| True | True | False | 2,3,4,8,9,12,17,18 |
| True | True | True | 2,3,4,8,9,10,17,18 |

# PCG Example

```
1  public void main() {
2      foo(1);
3      if (C1) {
4          foo(2);
5      } else {
6          foo(3);
7      }
8      if (C2) {
9          if (C3) {
10             foo(4);
11         } else {
12             crash();
13         }
14     } else {
15         crash();
16     }
17     foo(5);
18     return;
19 }
```

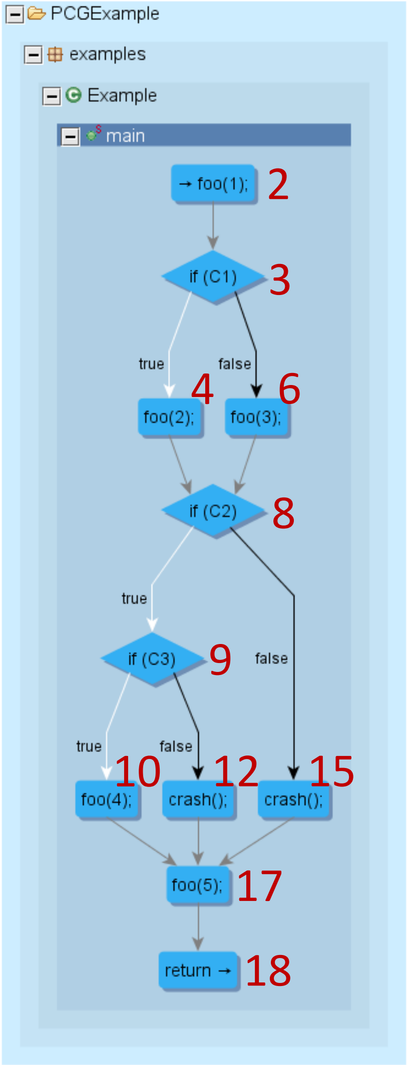


4 of 6 behaviors have “crash” events.
2 of 6 behaviors do not have “crash” events.

| C1 | C2 | C3 | Behavior |
|---|---|---|---|
| False | False | N/A | 2,3,6,8,15,17,18 |
| False | True | False | 2,3,6,8,9,12,17,18 |
| ~~False~~ | ~~True~~ | ~~True~~ | ~~2,3,6,8,9,10,17,18~~ |
| True | False | N/A | 2,3,4,8,15,17,18 |
| True | True | False | 2,3,4,8,9,12,17,18 |
| ~~True~~ | ~~True~~ | ~~True~~ | ~~2,3,4,8,9,10,17,18~~ |

# PCG Example

```
 1  public void main() {
 2      foo(1);
 3      if (C1) {
 4          foo(2);
 5      } else {
 6          foo(3);
 7      }
 8      if (C2) {
 9          if (C3) {
10              foo(4);
11          } else {
12              crash();
13          }
14      } else {
15          crash();
16      }
17      foo(5);
18      return;
19  }
```

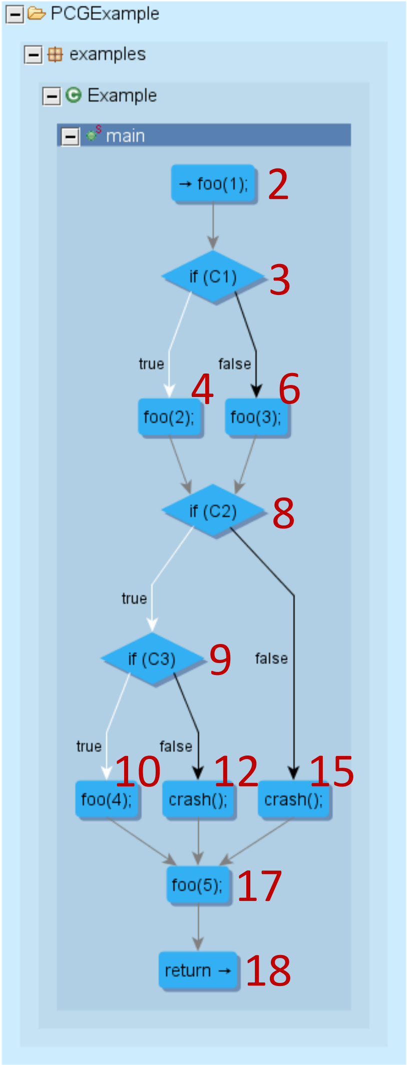


Consider necessary conditions for “crash” events.

| C1 | C2 | C3 | Behavior |
|---|---|---|---|
| False | False | N/A | 2,3,6,8,15,17,18 |
| False | True | False | 2,3,6,8,9,12,17,18 |
| ~~False~~ | ~~True~~ | ~~True~~ | ~~2,3,6,8,9,10,17,18~~ |
| True | False | N/A | 2,3,4,8,15,17,18 |
| True | True | False | 2,3,4,8,9,12,17,18 |
| ~~True~~ | ~~True~~ | ~~True~~ | ~~2,3,4,8,9,10,17,18~~ |

# PCG Example

```
 1  public void main() {
 2      foo(1);
 3      if (C1) {
 4          foo(2);
 5      } else {
 6          foo(3);
 7      }
 8      if (C2) {
 9          if (C3) {
10              foo(4);
11          } else {
12              crash();
13          }
14      } else {
15          crash();
16      }
17      foo(5);
18      return;
19  }
```



There are 3 unique behaviors w.r.t. “crash” events.

| C1 | C2 | C3 | Homomorphic Behavior |
|---|---|---|---|
| False | False | N/A | 8,15 |
| False | True | False | 8,9,12 |
| ~~False~~ | ~~True~~ | ~~True~~ | ~~8,9~~ |
| True | False | N/A | 8,15 |
| True | True | False | 8,9,12 |
| ~~True~~ | ~~True~~ | ~~True~~ | ~~8,9~~ |

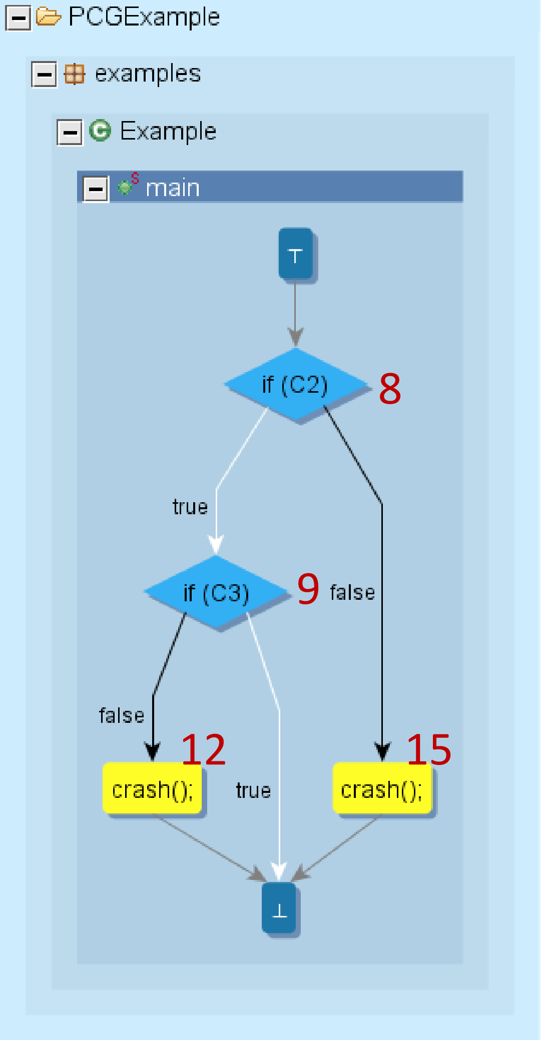
B1

B2

B3

# PCG Example

```
1  public void main() {
2      foo(1);
3      if (C1) {
4          foo(2);
5      } else {
6          foo(3);
7      }
8      if (C2) {
9          if (C3) {
10             foo(4);
11         } else {
12             crash();
13         }
14     } else {
15         crash();
16     }
17     foo(5);
18     return;
19 }
```



2 of 3 homomorphic behaviors (B1, B2) have "crash" events.
1 of 3 homomorphic behaviors (B3) are void w.r.t. "crash" events.

| C1 | C2 | C3 | Homomorphic Behavior | |
|---|---|---|---|---|
| False | False | N/A | 8,15 | B1 |
| False | True | False | 8,9,12 | B2 |
| ~~False~~ | ~~True~~ | ~~True~~ | ~~8,9~~ | B3 |
| True | False | N/A | 8,15 | |
| True | True | False | 8,9,12 | |
| ~~True~~ | ~~True~~ | ~~True~~ | ~~8,9~~ | |

# Demo

- Interactive Smart Views
- Automatic Pairing Analysis of Lock/Unlock events