EECS, Lassonde School of Engineering, York

Mysterious Code & Specifications

Jonathan S. Ostroff
Software Engineering Laboratory

Code Complexity

"Spaghetti code":
Incomprehensible code due to unnecessary coupling, jumps, gotos, or high complexity

Carnegie Mel

Toyota "Unintended Acceleration" Has Killed 89

A 2005 Toyota Prius, which was in an accident, is seen at a police station in Harrison, New York, Wednesday, March 10, 2010. The driver of the Toyota Prius told police that the car accelerated on its own, then lurched down a driveway, across a road and into a stone wall. (AP Photo/Seth Wenig) / **AP PHOTO/SETH WENIG**

Unintended acceleration in Toyota vehicles may have been involved in the deaths of 89 people over the past decade, upgrading the number of deaths possibly linked to the massive recalls, the government said Tuesday.

- McCabe Cyclomatic Complexity metric
  - Number of "eyes" in flow control graph
  - Unit tests harder with complex graph
    - Over 50 is considered "untestable"
- Toyota ETCS code:
  - 67 functions with complexity over 50
  - **Throttle angle function complexity = 146;** 1300 lines long, no unit test plan
    [Bookout 2013-10-14 31:10-32:23; 32:15-23]

ASSOCIATED PRESS
BY ASSOCIATED PRESS | April 2, 2018, 7:45AM

Tesla says Autopilot self-driving system engaged in fatal Bay Area crash

NEW YORK — The vehicle in a fatal March 23 crash in the San Francisco Bay Area was operating on Autopilot, making it the latest accident to involve a semi-autonomous vehicle, Tesla confirmed.

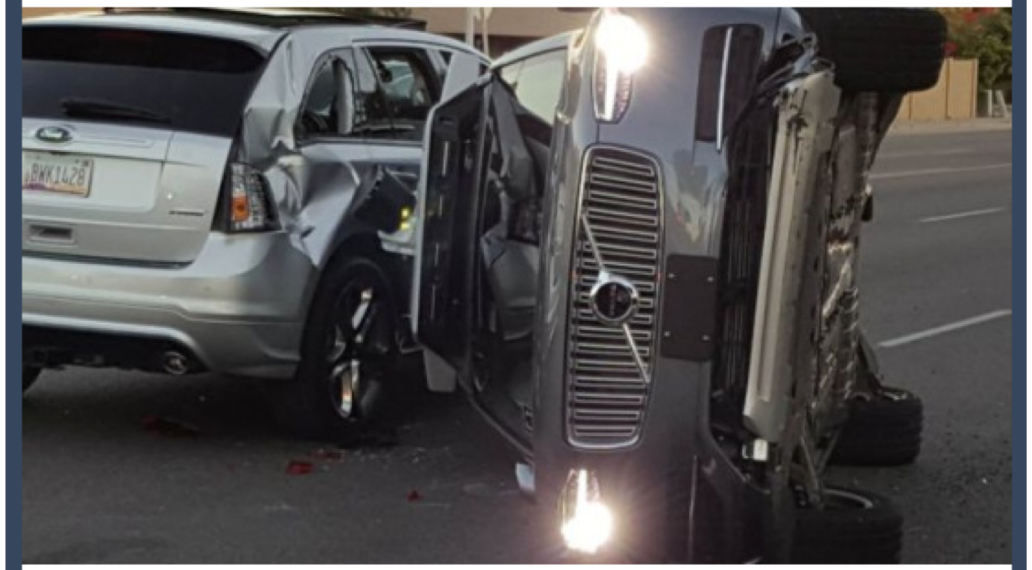
Uber suspends self-driving cars after Arizona crash

Uber suspends self-driving cars after Arizona crash

BBC 28 March 2017

BBC

http://www.bbc.com/news/technology-39397211



A self-driven Volvo SUV owned and operated by Uber Technologies Inc. is flipped on its side after a collision in Tempe, Arizona, U.S. on March 24, 2017.

Uber said the car was in self-driving mode at the time of the crash
Uber has pulled its self-driving cars from the roads after an accident which left one of the vehicles on its side.

```
# mystery routine
def even(number):
        return number % 2 == 0
def odd(number):
        return number % 2 != 0
a = 4
b = 5
x = a
y = b
z = 0
while (y > 0 and even(y)) or odd(y):
        if y > 0 and even(y):
                x, y = x + x, y // 2
        else:
                y, z = y - 1, z + x
print(z)
```

```
q(a, b: INTEGER): INTEGER
        -- mystery routine?
    local
        x, y, z: INTEGER
    do
        from
            x := a; y := b; z := 0
        until
            not (y > 0 and even(y)) and not odd(y)
        loop
            if y > 0 and even(y) then
                y := y // 2; x := x + x
            else -- odd(y)
                y := y - 1; z := z + x
            end
        end
        Result := z
    end
```

Here is more information. What does this routine do?

```
q(a, b: INTEGER): INTEGER
        --comment??
    require
        b >= 0
    local
        x, y, z: INTEGER
    do
        from
            x := a; y := b; z := 0
        invariant
            ????

        until
            not (y > 0 and even(y)) and not odd(y)
        loop
            if y > 0 and even(y) then
                y := y // 2; x := x + x
            else
                check odd(y) end
                y := y - 1; z := z + x
            end
        variant
            ????
        end
        Result := z
    ensure
        ????
    end
```

```
q(a, b: INTEGER): INTEGER
        --comment??
    require
        b >= 0
    local
        x, y, z: INTEGER
    do
        from
            x := a; y := b; z := 0
        invariant
            ????

        until
            not (y > 0 and even(y)) and not odd(y)
        loop
            if y > 0 and even(y) then
                y := y // 2; x := x + x
            else
                check odd(y) end
                y := y - 1; z := z + x
            end
        variant
            ????
        end
        Result := z
    ensure
        ????
    end
```

```
q(a, b: INTEGER): INTEGER
        --comment??
    require
        b >= 0
    local
        x, y, z: INTEGER
    do
        from
            x := a; y := b; z := 0
        invariant
            ????
        until
            not (y > 0 and even(y)) and not odd(y)
        loop
            if y > 0 and even(y) then
                y := y // 2; x := x + x
            else
                check odd(y) end
                y := y - 1; z := z + x
            end
        variant
            ????
        end
        Result := z
    ensure
        ????
    end
```

```
q(a, b: INTEGER): INTEGER
        -- Result is `a' multiplied by `b'
    require
        b >= 0
    local
        x, y, z: INTEGER
    do
        from
            x := a; y :=
        invariant
            y >= 0
            z + x*y = a*b
        until
            not (y > 0 an
        loop
            if y > 0 and even(y) then
                y := y // 2; x := x + x
            else
                check odd(y) end
                y := y - 1; z := z + x
            end
        variant
            y
        end
        Result := z
    ensure
        Result = a*b
    end
```

```
q (a, b: INTEGER_32): INTEGER_32
        -- Result is `a' multiplied by `b'
    require
        b >= 0
    ensure
        Result = a * b
```

Equivalent to y=0

```
q(a, b: INTEGER): INTEGER
        -- Result is `a' multiplied by `b'
    require
        b >= 0
    local
        x, y, z: INTEGER
    do
        from
            x := a; y := b; z := 0
        invariant
            y >= 0
            z + x*y = a*b
        until
            not (y > 0 and even(y)) and not odd(y)
        loop
            if y > 0 and even(y) then
                y := y // 2; x := x + x
            else
                check odd(y) end
                y := y - 1; z := z + x
            end
        variant
            y
        end
        Result := z
    ensure
        Result = a*b
    end
```

```
q (a, b: INTEGER_32): INTEGER_32
        -- Result is `a' multiplied by `b'
    require
            b >= 0
    ensure
            Result = a * b
```

- Some embedded CPUs do not have a multiplier
- Also needed for arbitrary precision multiplication on 32 bit processors
- a*b = a + a + a … linear in b
- Whereas the q algorithm does it in log(b) time

Weakest Precondition Calculus

wp("x:=e",R) ≙ WD(e)∧R[x:=e] -- also simultaneous assign
-- WD(e) means expression e is well defined

wp("S1;S2") ≙ wp("S1",wp("S2",R)

wp("if B then S1 else S2",R)
≙ WD(B) ∧(B ⇒ wp("S1",R)) ∧(¬B ⇒ wp("S2",R))

x:=a; y:=b; z:=0 can be reduced to the simultaneous assignment:
x, y, z := a, b, 0

{Q} program {R} ≡ Q ⇒ wp("program",R)

Weakest Precondition Calculus

```
r(a:T) -- routine r
 require Q
 do
    from init
    invariant I
    until B
    do
       body
    variant V
    end
 ensure R
 end
```

Proof Obligations for loop

1. {Q} init {I}
2. {I∧¬B} body {I}
3. (I∧B) ⇒ R
4. (I∧¬B) ⇒ (V >= 0)
5. {I∧¬B ∧ V=V_0} body {V<V_0}

Note: This also means that:

V < 0 implies (not I) or B

(i.e. we have terminated)

(1) Q ⇒ wp("x, y, z := a, b, 0", I) ✓

I: y ≥ 0 ∧ z + x*y = a*b

```
  wp("x, y, z := a, b, 0", I)
= « definition of wp for assignment »
  I [x, y, z := a, b, 0]
= « substitution »
  b ≥ 0 ∧ 0 + a*b = a*b
= « arithmetic »
  b ≥ 0
= « definition of Q»
  Q
```

```
q(a, b: INTEGER): INTEGER
        -- Result is `a' multiplied by `b'
    require
        b >= 0
    local
        x, y, z: INTEGER
    do
        from
            x := a; y := b; z := 0
        invariant
            y >= 0
            z + x*y = a*b
        until
            not (y > 0 and even(y)) and not odd(y)
        loop
            if y > 0 and even(y) then
                y := y // 2; x := x + x
            else
                check odd(y) end
                y := y - 1; z := z + x
            end
        variant
            y
        end
        Result := z
    ensure
        Result = a*b
    end
```

$$\{I \wedge \neg B \wedge B1\}\ S1\ \{I\}$$
$$= \{I \wedge B1\}\ S1\ \{I\}$$

$$I: y \geq 0 \wedge z + x*y = a*b$$

```
  wp("x := x + x ‖ y := y÷2", I)
= « definition of wp for assignment »
  I[x := x + x ‖ y := y ÷ 2]
= « substitution »
  y÷2≥0 ∧ z + (x+x)*(y÷2) = a*b
= « assume B1: y > 0 ∧ even(y), thus by
     arithmetic: (x+x)*(y÷2) = 2x*(y÷2) = x*y »
  y ≥ 0 ∧ z + x*y = a*b
= « defn. of I »
  I
```

Thus B1 ⇒ (I ≡ wp("S1", I),
i.e. {I ∧ B1} S1 {I}

```
q(a, b: INTEGER): INTEGER
        -- Result is `a' multiplied by `b'
    require
        b >= 0
    local
        x, y, z: INTEGER
    do
        from
            x := a; y := b; z := 0
        invariant
            y >= 0
            z + x*y = a*b
        until
            not (y > 0 and even(y)) and not odd(y)
        loop
            if y > 0 and even(y) then
                y := y // 2; x := x + x
            else
                check odd(y) end
                y := y - 1; z := z + x
            end
        variant
            y
        end
        Result := z
    ensure
        Result = a*b
    end
```

$$\{I \wedge \neg B \wedge B2\}\ S2\ \{I\}$$
$$= \{I \wedge B2\}\ S2\ \{I\}$$

$$I: y \geq 0 \wedge z + x*y = a*b$$

wp("y := y-1 ‖ z := z+x", I)
= « definition of wp for assignment »
I[y := y-1 ‖ z := z+x]
= « substitution »
(y-1≥0) ∧ (z+x + x*(y-1) = a*b)
= « arithmetic: z+x + x*(y-1) = z + x*y»
(y-1≥0) ∧ (z + x*y = a*b)
= « assume B2: odd(y), i.e. y≠0:
arithmetic: y≠0∧(y-1≥0) ≡ y≥0»
y≥0 ∧ (z + x*y = a*b)

Thus, B2 ⇒ (I ≡ wp("S2", I)),
i.e. I ∧ B2 ⇒ wp("S2", R), i.e. {I ∧ B2} S2 {I}

```
q(a, b: INTEGER): INTEGER
        -- Result is `a' multiplied by `b'
    require
        b >= 0
    local
        x, y, z: INTEGER
    do
        from
            x := a; y := b; z := 0
        invariant
            y >= 0
            z + x*y = a*b
        until
            not (y > 0 and even(y)) and not odd(y)
        loop
            if y > 0 and even(y) then
                y := y // 2; x := x + x
            else
                check odd(y) end
                y := y - 1; z := z + x
            end
        variant
            y
        end
        Result := z
    ensure
        Result = a*b
    end
```

```
 5. {P ∧ Bi ∧ t=T} Si {t < T}
==============================
 invariant P: y ≥ 0 ∧ z + x*y = a*b
 B1: y > 0 ∧ even(y)
 variant t: y
 Thus: {P ∧ B1 ∧ y=T} S1 {y < T}

   wp("x := x + 2 || y := y ÷ 2", y < T)
 = « substitution »
   y ÷ 2 < T
 ⇐ « arithmetic: y ÷ 2 < y »
   y = T
 ⇐ « strengthening with P ∧ B1 »
  P ∧ B1 ∧ y = T

Thus:  (P ∧ B1 ∧ y = T) ⇒ wp("x := x+2 || y := y÷2", y<T)
Hence: {P ∧ B1 ∧ t=T} S1 {t < T}
```

variant

dafny

Microsoft Research

about Dafny - A language and program verifier for functional correctness

Dafny is a programming language with imperative and functional features, as well as specification constructs for describing intended behavior. The Dafny verifier checks that programs live up to their specifications.

Is this program correct?

```
1 function Fibonacci(n: int): int
2   decreases n
3 {
4   if n < 2 then n else Fibonacci(n+2) + Fibonacci(n+1)
5 }
6
```

		Description	Line	Column
⊗	1	failure to decrease termination measure	4	23

```
Dafny 2.1.1.10209
stdin.dfy(4,23): Error: failure to decrease terminatior

Dafny program verifier finished with 0 verified, 1 errc
```

Industrial strength tools can now automate verification

Dijkstra's non-deterministic guarded command language

```
do  B1  →  S1
[]  B2  →  S2
od
```

BB = B1 V B2

So long as BB holds,
choose a guarded command
Bi→ Si
and execute it

Dijkstra's non-deterministic guarded command language

```
do  B1  →  S1
[]  B2  →  S2
od
```

BB = B1 V B2

So long as BB holds,
choose a guarded command
Bi→ Si
and execute it

Dijkstra's guarded command language

$$BB = B1 \vee B2$$

$$\{P\}$$
$$\textbf{do } \{P \wedge B1\}\ B1 \rightarrow S1\ \{P\}$$
$$[]\ \{P \wedge B2\}\ B2 \rightarrow S2\ \{P\}$$
$$\textbf{od}$$
$$\{P \wedge \neg BB\}$$

The algorithm in Dijkstra's guarded command language

```
BB = B1 ∨ B2


x, y, z := a, b, 0


do y > 0 ∧ even(y) → x := x + x ‖ y := y ÷ 2
[] odd(y)          → y := y - 1 ‖ z := z + x
od
```

The algorithm in Dijkstra's guarded command language

```
BB = B1 ∨ B2


{Q: b ≥ 0}
x, y, z := a, b, 0
{P: y ≥ 0 ∧ z + x*y = a*b}
{variant t: y}
do y > 0 ∧ even(y) → {P ∧ B1} x := x + x ‖ y := y ÷ 2 {P}
[] odd(y)          → {P ∧ B2} y := y - 1 ‖ z := z + x {P}
od
{P ∧ ¬BB}
{R: z = a*b}
```

Dijkstra's guarded command language

```
BB = B1 ∨ B2

{Q}
init
{invariant: P}
{variant: t}
do {P ∧ B1} B1 → S1 {P}
[] {P ∧ B2} B2 → S2 {P}
od
{P ∧ ¬BB}
{R}
```

Prove {Q} loop {R}

1. Prove that P is true initially, Q ⇒ P
2. Prove that each guarded command preserves P, i.e. {P ∧ Bi} Si {P}
3. Prove that the invariant entails the postcondition, i.e. P ∧ ¬ B ⇒ R
4. Show that variant is bounded below
   P ∧ BB ⇒ t >= 0
5. Show that the variant decreases each iteration
   {P ∧ Bi ∧ t=T} Si {t < T}