CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
SYMBOLIC TESTING

Autumn 2011

Testing

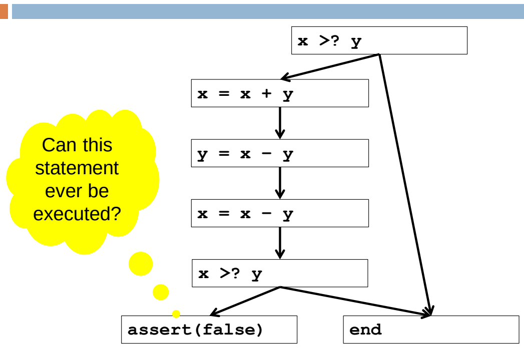
Simple program – what do we want to know about it?

Not a new question for us – let's consider it with white-box testing

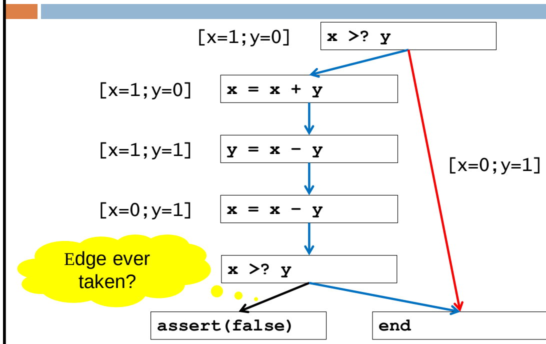
```
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert(false)
}
```

Visser, Pasareanu & Mehlitz

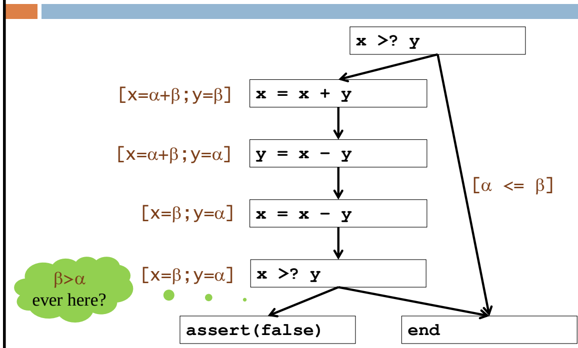
Control flow graph (CFG)



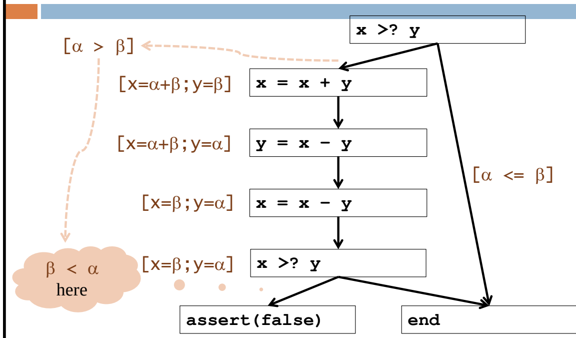
Edge coverage



Symbolic execution [x=α;y=β]



Symbolic execution

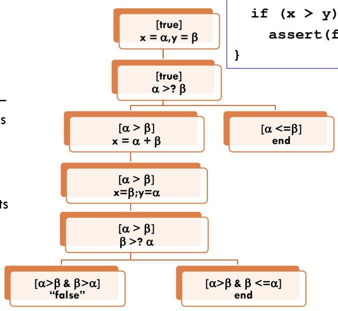


What's really going on?

7

```
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert(false)
}
```

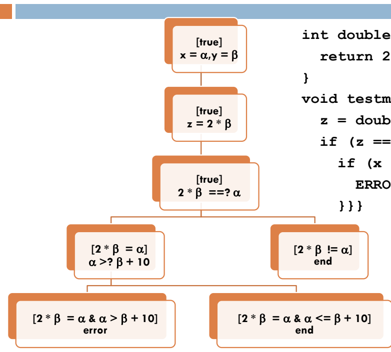
- Create a symbolic execution tree
- Explicitly track path conditions
- Solve path conditions – "how do you get to this point in the execution tree?" – to define test inputs
- Goal: define test inputs that reach all reachable statements



Another example (Sen and Agha)

8

```
int double (int v){
  return 2*v;
}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
}}}
```



Error: possible by solving equations

9

$[2 * \beta = \alpha \ \& \ \alpha > \beta + 10]$

$\equiv [2 * \beta > \beta + 10]$

$\equiv [\beta > 10]$

$\equiv [\beta > 10 \ \& \ 2 * \beta = \alpha]$

Way cool – we're done!

10

- First example can't reach **`assert(false)`**, and it's easy to reach **`end`** via both possible paths
- Second example: can reach **`error`** and **`end`** via both possible paths
- Well, what if we can't solve the path conditions?
  - Some arithmetic, some recursion, some loops, some pointer expressions, etc.
  - We'll see an example
- What if we want specific test cases?

Concolic testing: Sen et al.

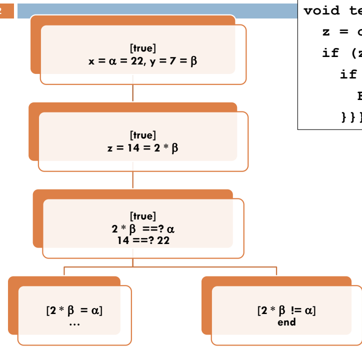
11

- Basically, combine concrete and symbolic execution
- More precisely…
  - Generate a random concrete input
  - Execute the program on that input both concretely and symbolically simultaneously
  - Follow the concrete execution and maintain the path conditions along with the corresponding symbolic execution
  - Use the path conditions collected by this guided process to constrain the generation of inputs for the next iteration
  - Repeat until test inputs are produced to exercise all feasible paths

2nd example redux
1st iteration x=22, y=7

12

```
int double (int v){
  return 2*v;
}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
}}}
```

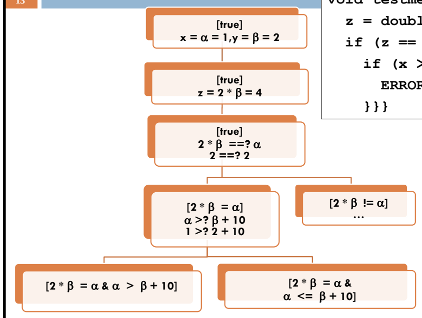


- **Now solve $2 * \beta = \alpha$ to force the other branch**
- **`x = 1; y = 2` is one solution**

2^{nd} example
2^{nd} iteration x=1, y=2

13

```
int double (int v){
  return 2*v;
}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
}}}
```

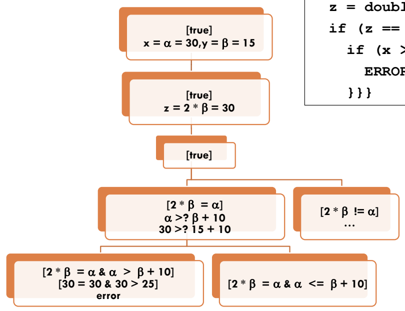


- Now solve $2 * \beta = \alpha$ & $\alpha <= \beta + 10$ to force the other branch
- `x = 30;` `y = 15` is one solution

2^{nd} example
3^{rd} iteration x=30, y=15

14

```
int double (int v){
  return 2*v;
}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
}}}
```



- Now solve $2 * \beta = \alpha$ & $\alpha <= \beta + 10$ to force the other branch
- `x = 30; y = 15` is one solution

Three concrete test cases

15

```
int double (int v){ return 2*v;}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
    }
  }
}
```

x	y	
22	7	Takes first else
2	1	Takes first then and second else
30	15	Takes first and second then

Concolic testing example
P. Sağlam

- Random seed
  - `x = -3; y = 7`
- Concrete
  - `z = 9`
- Symbolic
  - $z = x^3+3x^2+9$
- Take **then** branch with constraint $x^3+3x^2+9 \;!= y$

```
void test_me(int x,int y){
  z = x*x*x + 3*x*x + 9;
  if(z != y){
    printf("Good branch");
  } else {
    printf("Bad branch");
    abort();
  }
}
```

- Take **else** branch with constraint $x^3+3x^2+9 = y$

Concolic testing example
P. Sağlam

- Solving is hard for $x^3+3x^2+9 = y$
- So use **z**'s concrete value, which is currently 9, and continue concretely
- `9 != 7` so **then** is good
- Symbolically solve `9 = y` for **else** clause
- Execute next run with `x = -3; y = 9` so **else** is **bad**

```
void test_me(int x,int y){
  z = x*x*x + 3*x*x + 9;
  if(z != y){
    printf("Good branch");
  } else {
    printf("Bad branch");
    abort();
  }
}
```

- When symbolic expression becomes unmanageable (e.g., non-linear) replace it by concrete value

Concolic testing example
P. Sağlam

- Random memory graph reachable from **p**
- Random value for **x**
- Probability of reaching `abort( )` is extremely low

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Let's try it

19

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x)
    {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete	Symbolic	Constraints
p=NULL; x=236		

Let's try it

20

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x)
    {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete	Symbolic	Constraints
p=[634,NULL]; x=236		

Let's try it

21

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x)
    {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete	Symbolic	Constraints
p=[3,p]; x=1		

Let's try it

22

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x)
    {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete	Symbolic	Constraints

Concolic: status

23

- The jury is still out on concolic testing – but it surely has potential
- There are many papers on the general topic
- Here's one that is somewhat high-level Microsoft-oriented
  - Godefroid et al. Automating Software Testing Using Program Analysis *IEEE Software* (Sep/Oct 2008)
  - They tend to call the approach DART – Dynamic Automated Random Testing

Next steps

24

- Worksheets