Programm- & Systemverifikation

Assertions & Testing: Exercises

Georg Weissenbacher

184.741

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – *incorrect* state that may lead to failure
  - **Failure** – deviation from *desired* behaviour
- We specified *intended* behaviour using **assertions**
- We proved our programs correct (inductive invariants).
- Coverage Metrics tell us when to stop testing.
- Heard about Automated Test-Case Generation.

In this Lecture

More Examples and Exercises for

- Bugs
- Assertions
- Testing
- Test Case Generation
- Inductive Invariants

Spot the Bug

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[HeartbeatMessage.payload_length];
  opaque padding[padding_length];
} HeartbeatMessage;
/* ... */
/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload); /* puts 2 bytes of p into payload */
p1 = p;
/* ... */
if (hbtype == TLS1_HB_REQUEST) {
  unsigned char *buffer, *bp;
  int r;
  buffer = OPENSSL_malloc(1+2+payload+padding);
  bp = buffer;
  *bp++ = TLS1_HB_RESPONSE;
  s2n(payload, bp); /* puts 16-bit value into bp */
  memcpy(bp, p1, payload);
  r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer,
      3+payload+padding);
}
```

Heartbleed OpenSSL bug

- TLS heartbeat mechanism keeps connections alive
  - receiver *must* send a corresponding response carrying an exact copy of the payload of the received request
- `payload` is trusted without bounds check
- attacker can request slice of memory up to 2^{16} bytes, obtain
  - long-term server private keys
  - TLS session keys
  - confidential data like passwords
  - session ticket keys
- affected version: OpenSSL 1.01 through 1.01f

Assertions as formal specifications

- Assume:

  ```
  unsigned isqrt (unsigned x)
  ```

  computes largest *integer* square root of `x`
- Write assertion that fails if result is wrong!

Assertions as formal specifications

- Assume:

  ```
  unsigned isqrt (unsigned x)
  ```

  computes largest *integer* square root of `x`
- Write assertion that fails if result is wrong!

```
unsigned r = isqrt (x);
assert (r*r <= x && x <= (r+1)*(r+1));
```

Assertions as formal specifications

- Assume:

  ```
  unsigned isqrt (unsigned x)
  ```

  computes largest *integer* square root of `x`
- Write assertion that fails if result is wrong!

```
unsigned r = isqrt (x);
assert (r*r <= x && x <= (r+1)*(r+1));
```

- Note: Assertion doesn’t tell us how `isqrt` works!

Assertions as formal specifications

- Assume:

  ```
  unsigned gcd (unsigned x, unsigned y)
  ```

  computes *greatest common divisor* of `x` and `y`
- Write assertion that fails if result is wrong!

  ```
  unsigned r = gcd (x, y);
  ...
  ```

Assertions as formal specifications

```
unsigned r = gcd (x, y);
...
```

What are the properties of the greatest common divisor `r`?

Assertions as formal specifications

```
unsigned r = gcd (x, y);
...
```

What are the properties of the greatest common divisor `r`?

- `(x % r == 0) && (y % r == 0)`

Assertions as formal specifications

```
unsigned r = gcd (x, y);
assert ((x % r == 0) && (y % r == 0));
```

What are the properties of the greatest common divisor `r`?

- `(x % r == 0) && (y % r == 0)`

Assertions as formal specifications

```
unsigned r = gcd (x, y);
assert ((x % r == 0) && (y % r == 0));
```

What are the properties of the greatest common divisor `r`?

- `(x % r == 0) && (y % r == 0)`
- Is this *sufficient?*

Assertions as formal specifications

```
unsigned r = gcd (x, y);
assert ((x % r == 0) && (y % r == 0));
```

What are the properties of the greatest common divisor `r`?

- `(x % r == 0) && (y % r == 0)`
- Is this *sufficient?*
    - What if `gcd (12, 36)` returns 3?

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
```

Properties of `r` (for `r = gcd(x, y)`)

- `IS_CD (r, x, y)`
- $\nexists t \in \mathbb{N} \,.\, \texttt{IS\_CD}(t, x, y) \wedge (t > \texttt{r})$

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
```

Properties of `r` (for `r = gcd(x, y)`)

- `IS_CD (r, x, y)`
- $\nexists t \in \mathbb{N} \,.\, \texttt{IS\_CD}(t, x, y) \wedge (t > \texttt{r})$
  - C++ doesn't have quantifiers
  - $\mathbb{N}$ has infinitely many elements

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
```

Properties of `r` (for `r = gcd(x, y)`)

- `IS_CD (r, x, y)`
- $\nexists t \in \mathbb{N} \,.\, \texttt{IS\_CD}(t, x, y) \land (t > \texttt{r})$
  - C++ doesn't have quantifiers
  - $\mathbb{N}$ has infinitely many elements
  - What else do we know about `%`?

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
```

Properties of `r` (for `r = gcd(x, y)`)

- `IS_CD (r, x, y)`
- $\nexists t \in \mathbb{N} \,.\, \texttt{IS\_CD}(t, x, y) \land (t > \texttt{r})$
  - C++ doesn't have quantifiers
  - $\mathbb{N}$ has infinitely many elements
  - What else do we know about `%`?
- $(\texttt{r} > \texttt{y}) \Rightarrow (\texttt{y}\%\texttt{r} = \texttt{y})$

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
```

Properties of `r` (for `r = gcd(x, y)`)

- `IS_CD (r, x, y)`
- $\nexists t \in \mathbb{N} \,.\, \mathtt{IS\_CD}(t, x, y) \land (t > \mathtt{r})$
  - C++ doesn't have quantifiers
  - $\mathbb{N}$ has infinitely many elements
  - What else do we know about %?
- $(\mathtt{r} > \mathtt{y}) \Rightarrow (\mathtt{y\%r} = \mathtt{y})$
  - therefore, $\mathtt{r} \leq \min(x, y)$

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
```

Properties of `r` (for `r = gcd(x, y)`)

- `IS_CD (r, x, y)`
- $\nexists t \in \mathbb{N} \,.\, \mathtt{IS\_CD}(t, x, y) \wedge (t > \mathtt{r}) \wedge (t \leq \min(x, y))$
  - C++ doesn't have quantifiers
  - $\mathbb{N}$ has infinitely many elements
  - What else do we know about %?
- $(\mathtt{r} > \mathtt{y}) \Rightarrow (\mathtt{y}\%\mathtt{r} = \mathtt{y})$
  - therefore, $\mathtt{r} \leq \min(x, y)$

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
#define min(x, y) (((x)<(y))?(x):(y))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
```

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
#define min(x, y) (((x)<(y))?(x):(y))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
assert (∄t ∈ ℕ . IS_CD(t, x, y) ∧ (t > r) ∧ (t ≤ min(x, y)));
```

- What about the quantifier?

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
#define min(x, y) (((x)<(y))?(x):(y))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
```

$\mathtt{assert}\ (\nexists t \in \mathbb{N}\,.\,\mathtt{IS\_CD}(t, x, y) \wedge (t > \mathtt{r}) \wedge (t \leq \mathrm{min}(x, y)))\,;$

- What about the quantifier?
  - $\mathtt{r} < t \leq \mathrm{min}(x, y)$, we can use a loop!

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
#define min(x, y) (((x)<(y))?(x):(y))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
for (unsigned t=r+1; t <= min(x, y); t++)
  assert (!IS_CD(t, x, y));
```

- Does not make assumptions about implementation

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
#define min(x, y) (((x)<(y))?(x):(y))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
for (unsigned t=r+1; t <= min(x, y); t++)
  assert (!IS_CD(t, x, y));
```

- Does not make assumptions about implementation
- Admittedly, not very efficient
  - Only for testing!
  - Turn it off in release version.

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
#define min(x, y) (((x)<(y))?(x):(y))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
for (unsigned t=r+1; t <= min(x, y); t++)
  assert (!IS_CD(t, x, y));
```

- This specification is not *executable*
- But very close to full-blown (inefficient) implementation

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
#define min(x, y) (((x)<(y))?(x):(y))
unsigned r = gcd (x, y);
assert (IS_CD(r, x, y));
for (unsigned t=r+1; t <= min(x, y); t++)
  assert (!IS_CD(t, x, y));
```

- This specification is not *executable*
- But very close to full-blown (inefficient) implementation
  - We can implement a “prototype”

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
#define min(x, y) (((x)<(y))?(x):(y))

unsigned gcd (x, y) {
  for (unsigned t = min(x, y); t > 0; t--) {
    if (IS_CD(t, x, y))
      return t;
  }

}
```

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
#define min(x, y) (((x)<(y))?(x):(y))

unsigned gcd (x, y) {
  for (unsigned t = min(x, y); t > 0; t--) {
    if (IS_CD(t, x, y))
      return t;
  }

}
```

- Wait, can we reach end of function without return?

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
#define min(x, y) (((x)<(y))?(x):(y))
#define max(x, y) (((x)<(y))?(y):(x))
unsigned gcd (x, y) {
  for (unsigned t = min(x, y); t > 0; t--) {
    if (IS_CD(t, x, y))
      return t;
  }
  return max(x, y);
}
```

- Wait, can we reach end of function without return?
  - Yes, if $\mathtt{min}(\mathtt{x}, \mathtt{y}) = 0$
  - In this case, return `max(x, y)` (since $\mathtt{gcd}(0, \mathtt{x}) = \mathtt{x}$)

Assertions as formal specifications

```
#define IS_CD(r, x, y) (((x)%(r)==0) && ((y)%(r)==0))
#define min(x, y) (((x)<(y))?(x):(y))
#define max(x, y) (((x)<(y))?(y):(x))
unsigned gcd (x, y) {
  for (unsigned t = min(x, y); t > 0; t--) {
    if (IS_CD(t, x, y))
      return t;
  }
  return max(x, y);
}
```

- This implementation is inefficient!
    - But we can use it as a prototype!

```c
char is_cd (unsigned r, unsigned x, unsigned y) {
  return ((x % r == 0) && (y % r == 0));
}

unsigned gcd_proto (unsigned x, unsigned y) {
  unsigned t = min (x, y);
  for (; t > 0; t--) {
    if (is_cd (t, x, y))
      return t;
    }
  return max (x, y);
}
```

Euclid’s Algorithm

```
unsigned gcd_impl (unsigned x, unsigned y)
{
  unsigned k = x;
  unsigned m = y;

  while (k != m) {
    if (k > m) {
      k = k - m;
    }
    else {
      m = m - k;
    }
  }
  return k;
}
```

Euclid’s Algorithm

```
unsigned gcd_impl (unsigned x, unsigned y)
{
  unsigned k = x;
  unsigned m = y;

  while (k != m) {
    if (k > m) {
      k = k - m;
    }
    else {
      m = m - k;
    }
  }
  return k;
}
```

- Why does this work?

Euclid’s Algorithm: Correctness

```
unsigned k = x;
unsigned m = y;

while (k != m) {
  if (k > m)  k = k - m;
  else m = m - k;
}
return k;
```

Properties of `gcd`:

- If `x` $=$ `y`, then `gcd (x,y)` = `gcd (x,x)` = `x`
- If `x` $>$ `y`, then `gcd (x,y)` = `gcd (x-y,y)`

Euclid’s Algorithm: Correctness

If $\mathtt{x} > \mathtt{y}$, then `gcd (x,y)` = `gcd (x-y,y)`. Proof:

- Suppose `IS_CD(r, x, y)`. Then

  $$\exists \mathtt{n}, \mathtt{m} \,.\, (\mathtt{x} = \mathtt{n} \cdot \mathtt{r}) \wedge (\mathtt{y} = \mathtt{m} \cdot \mathtt{r})$$

  Therefore,

  $$\mathtt{x} - \mathtt{y} = \mathtt{n} \cdot \mathtt{r} - \mathtt{m} \cdot \mathtt{r} = (\mathtt{n} - \mathtt{m}) \cdot \mathtt{r}$$

  and thus $((\mathtt{x} - \mathtt{y}) \% \mathtt{r}) = 0$.

Euclid's Algorithm: Correctness

If $\mathtt{x} > \mathtt{y}$, then `gcd (x,y)` = `gcd (x-y,y)`. Proof:

- Suppose `IS_CD(r, x, y)`. Then

  $$\exists \mathtt{n}, \mathtt{m} \,.\, (\mathtt{x} = \mathtt{n} \cdot \mathtt{r}) \wedge (\mathtt{y} = \mathtt{m} \cdot \mathtt{r})$$

  Therefore,

  $$\mathtt{x} - \mathtt{y} = \mathtt{n} \cdot \mathtt{r} - \mathtt{m} \cdot \mathtt{r} = (\mathtt{n} - \mathtt{m}) \cdot \mathtt{r}$$

  and thus $((\mathtt{x} - \mathtt{y}) \% \mathtt{r}) = 0$.
- Using similar reasoning, we can also show that

  $$\mathtt{IS\_CD}(\mathtt{r}, \mathtt{x} - \mathtt{y}, \mathtt{y}) \Rightarrow \mathtt{IS\_CD}(\mathtt{r}, \mathtt{x}, \mathtt{y}).$$

Euclid's Algorithm: Correctness

If $\mathtt{x} > \mathtt{y}$, then `gcd (x,y) = gcd (x-y,y)`. Proof:

- Suppose `IS_CD(r, x, y)`. Then

  $$\exists \mathtt{n}, \mathtt{m} \,.\, (\mathtt{x} = \mathtt{n} \cdot \mathtt{r}) \wedge (\mathtt{y} = \mathtt{m} \cdot \mathtt{r})$$

  Therefore,

  $$\mathtt{x} - \mathtt{y} = \mathtt{n} \cdot \mathtt{r} - \mathtt{m} \cdot \mathtt{r} = (\mathtt{n} - \mathtt{m}) \cdot \mathtt{r}$$

  and thus $((\mathtt{x} - \mathtt{y})\%\mathtt{r}) = 0$.
- Using similar reasoning, we can also show that

  $$\mathtt{IS\_CD}(\mathtt{r}, \mathtt{x} - \mathtt{y}, \mathtt{y}) \Rightarrow \mathtt{IS\_CD}(\mathtt{r}, \mathtt{x}, \mathtt{y}).$$

- Therefore

  $$\{\mathtt{r} \mid \mathtt{IS\_CD}(\mathtt{r}, \mathtt{x}, \mathtt{y})\} = \{\mathtt{r} \mid \mathtt{IS\_CD}(\mathtt{r}, \mathtt{x} - \mathtt{y}, \mathtt{y})\}$$

Euclid's Algorithm: Correctness

If $\mathtt{x} > \mathtt{y}$, then `gcd (x,y)` = `gcd (x-y,y)`. Proof:

- Suppose `IS_CD(r, x, y)`. Then

  $$\exists \mathtt{n}, \mathtt{m} \,.\, (\mathtt{x} = \mathtt{n} \cdot \mathtt{r}) \wedge (\mathtt{y} = \mathtt{m} \cdot \mathtt{r})$$

  Therefore,

  $$\mathtt{x} - \mathtt{y} = \mathtt{n} \cdot \mathtt{r} - \mathtt{m} \cdot \mathtt{r} = (\mathtt{n} - \mathtt{m}) \cdot \mathtt{r}$$

  and thus $((\mathtt{x} - \mathtt{y}) \% \mathtt{r}) = 0$.
- Using similar reasoning, we can also show that

  $$\mathtt{IS\_CD}(\mathtt{r}, \mathtt{x} - \mathtt{y}, \mathtt{y}) \Rightarrow \mathtt{IS\_CD}(\mathtt{r}, \mathtt{x}, \mathtt{y}).$$

- Therefore

  $$\{\mathtt{r} \mid \mathtt{IS\_CD}(\mathtt{r}, \mathtt{x}, \mathtt{y})\} = \{\mathtt{r} \mid \mathtt{IS\_CD}(\mathtt{r}, \mathtt{x} - \mathtt{y}, \mathtt{y})\}$$

- In particular, the largest element in both sets is the same

Euclid’s Algorithm

```
unsigned gcd_impl (unsigned x, unsigned y)
{
  unsigned k = x;
  unsigned m = y;

  while (k != m) {
    if (k > m) {
      k = k - m;
    }
    else {
      m = m - k;
    }
  }
  return k;
}
```

Euclid’s Algorithm

```
unsigned gcd_impl (unsigned x, unsigned y)
{
  unsigned k = x;
  unsigned m = y;

  while (k != m) {
    if (k > m) {
      k = k - m;
    }
    else {
      m = m - k;
    }
  }
  return k;
}
```

- We can now use a Test Case Generator (e.g., KLEE)

Euclid’s Algorithm

- Let’s look at inputs `x=k=0`, `y=m=1`
  - What happens in this case?

```
while (k != m) {
  if (k > m) {
    k = k - m;
  }
  else {
    m = m - k;
  }
}
```

Euclid’s Algorithm

- Let’s look at inputs x=k=0, y=m=1
  - What happens in this case?

```
while (k != m) {
  if (k > m) {
    k = k - m;
  }
  else {
    m = m - k;
  }
}
```

- Number of loop iterations: ∞

Euclid’s Algorithm

```
unsigned gcd_impl(unsigned x, unsigned y)
{
  unsigned k = x;
  unsigned m = y;

  if ((x == 0) || (y == 0))
     return max (x, y);

  while (k != m) {
    if (k > m) {
      k = k - m;
    }
    else {
      m = m - k;
    }
  }
  return k;
}
```

Euclid's Algorithm

The program is correct; but not necessarily efficient!

- For 905 and 2, Euclid's algorithm loops 453 times

Euclid’s Algorithm

The program is correct; but not necessarily efficient!

- For 905 and 2, Euclid’s algorithm loops 453 times
- Maybe there is a more efficient algorithm?

Euclid’s Algorithm

The program is correct; but not necessarily efficient!

- For 905 and 2, Euclid’s algorithm loops 453 times
- Maybe there is a more efficient algorithm?
  - Euclid’s `gcd` deducts 2 from 905 452 times
  - 905 % 2 would yield the same result in one step!
  - Can also avoid $\texttt{k} > \texttt{m}$ comparison by swapping values!

Euclid’s Algorithm

```
unsigned gcd_impl2(unsigned x, unsigned y)
{
  unsigned k = max(x,y);
  unsigned m = min(x,y);

  while (m != 0) {
    unsigned r = k % m;
    k = m;
    m = r;
  }

  return k;
}
```

Euclid’s Algorithm

- Now the algorithm is much more efficient
- But are we pleased with these test cases?
    - What’s the coverage?

```c
#include <assert.h>
#define MIN(x, y) ((x)<(y))?(x):(y)
#define MAX(x, y) ((x)<(y))?(y):(x)

unsigned gcd (unsigned x, unsigned y)
{
  unsigned k = MAX (x,y);
  unsigned m = MIN (x,y);
  while (m != 0) {
    unsigned r = k % m;
    k = m; m = r;
  }
  return k;
}
int main(int argc, char** argv)
{
  assert (gcd (0,0)    == 0);
  assert (gcd (1,1)    == 1);
  assert (gcd (905,2)  == 1);
  assert (gcd (905,2)  == 1);
  assert (gcd (2,3)    == 1);
  assert (gcd (512,31) == 1);
}
```

GCOV Usage Revisited

- `gcc -g -fprofile-arcs -ftest-coverage -o gcd gcd.c`
  (use `clang` instead of `gcc` on newer Macs)
- `gcov -b gcd`
- `cat gcd.c.gcov`
- `./gcd ; gcov -b gcd`
- `cat gcd.c.gcov`

GCOV Results

```
function gcd called 6 returned 100% blocks executed 100%
        6:    5:unsigned gcd (unsigned x, unsigned y)
        -:    6:{
       18:    7:  unsigned k = MAX (x,y);
       18:    8:  unsigned m = MIN (x,y);
branch  0 taken 17%
branch  1 taken 83%
       23:    9:  while (m != 0) {
branch  0 taken 65%
branch  1 taken 35%
       11:   10:    unsigned r = k % m;
       11:   11:    k = m; m = r;
       11:   12:  }
        6:   13:  return k;
        -:   14:}
```

GCOV Results

Why is GCOV . . .

- reporting two branches?
  - Remember that the macros `MAX` and `MIN` both hide the same branch
- claiming that branch coverage hasn’t been reached?
  - `assert` is actually a macro, too.

Other Control-Flow-Based Coverage Metrics

- Test suite achieves full branch/decision coverage for `gcd`
- What about
  - condition coverage?
  - condition decision coverage?
  - MC/DC?
  - multiple condition coverage?

Other Control-Flow-Based Coverage Metrics

- Test suite achieves full branch/decision coverage for `gcd`
- What about
  - condition coverage?
  - condition decision coverage?
  - MC/DC?
  - multiple condition coverage?
- Only decisions in `gcd` are `(m != 0)` and `(x < y)`
  - Therefore, these notions coincide.

Data-Flow-Based Coverage Metrics

```
unsigned k, m;
if (x > y) {
  k = x; m = y
} else {
  k = y; m = x;
}
while (m != 0) {
  unsigned r = k % m;
  k = m; m = r;
}
return k;
```

x	y
0	0
1	1
905	2
2	3
512	31

- Do we achieve all-p-uses/some-c-uses coverage?
  (all definitions used, and if they affect decisions, then all affected decisions are executed)

How Can KLEE Generate Test Cases?

① Select a path in the function `gcd`

② Generate conditions depending on *symbolic* inputs

③ Find *satisfying assignment* (using SMT Solver)

④ Run Prototype on generated inputs

- Report generated inputs and output of oracle

④ If coverage reached, terminate; else goto ①

Automated Test-Case Generation

- E.g., want to cover `else`-branch at ①, loop at ② once

```
   unsigned k, m;
①  if (x > y) {
     k = x; m = y
   } else {
     k = y; m = x;
   }
②  while (m != 0) {
     unsigned r = k % m;
     k = m; m = r;
   }
   return k;
```

Automated Test-Case Generation

- E.g., want to cover `else`-branch at ①, loop at ② once

```
    unsigned k, m;
①   if (x > y) {
      k = x; m = y
    } else {
      k = y; m = x;
    }
②   while (m != 0) {
      unsigned r = k % m;
      k = m; m = r;
    }
    return k;
```

$\mathtt{x} \mapsto x_0, \mathtt{y} \mapsto y_0$

Automated Test-Case Generation

- E.g., want to cover `else`-branch at ①, loop at ② once

```
   unsigned k, m;
①  if (x > y) {
     k = x; m = y
   } else {
     k = y; m = x;
   }
②  while (m != 0) {
     unsigned r = k % m;
     k = m; m = r;
   }
   return k;
```

$\mathtt{x} \mapsto x_0, \mathtt{y} \mapsto y_0$

$(x_0 \leq y_0)$

Automated Test-Case Generation

- E.g., want to cover `else`-branch at ①, loop at ② once

```
   unsigned k, m;
①  if (x > y) {
     k = x; m = y
   } else {
     k = y; m = x;
   }
②  while (m != 0) {
     unsigned r = k % m;
     k = m; m = r;
   }
   return k;
```

$\mathtt{x} \mapsto x_0, \mathtt{y} \mapsto y_0$

$(x_0 \leq y_0)$

$\mathtt{k} \mapsto y_0, \mathtt{m} \mapsto x_0$

Automated Test-Case Generation

- E.g., want to cover `else`-branch at ①, loop at ② once

```
    unsigned k, m;
①   if (x > y) {
      k = x; m = y
    } else {
      k = y; m = x;
    }
②   while (m != 0) {
      unsigned r = k % m;
      k = m; m = r;
    }
    return k;
```

$\mathtt{x} \mapsto x_0, \mathtt{y} \mapsto y_0$

$(x_0 \leq y_0)$

$\mathtt{k} \mapsto y_0, \mathtt{m} \mapsto x_0$

$(x_0 \neq 0)$

Automated Test-Case Generation

- E.g., want to cover `else`-branch at ①, loop at ② once

	`unsigned k, m;`	$\mathtt{x} \mapsto x_0, \mathtt{y} \mapsto y_0$
①	`if (x > y) {`	$(x_0 \leq y_0)$
	`  k = x; m = y`	
	`} else {`	
	`  k = y; m = x;`	$\mathtt{k} \mapsto y_0, \mathtt{m} \mapsto x_0$
	`}`	
②	`while (m != 0) {`	$(x_0 \neq 0)$
	`  unsigned r = k % m;`	$\mathtt{r} \mapsto (y_0 \,\%\, x_0)$
	`  k = m; m = r;`	
	`}`	
	`return k;`	

Automated Test-Case Generation

- E.g., want to cover `else`-branch at ①, loop at ② once

	`unsigned k, m;`	$\mathtt{x} \mapsto x_0, \mathtt{y} \mapsto y_0$
①	`if (x > y) {`	$(x_0 \leq y_0)$
	`  k = x; m = y`	
	`} else {`	
	`  k = y; m = x;`	$\mathtt{k} \mapsto y_0, \mathtt{m} \mapsto x_0$
	`}`	
②	`while (m != 0) {`	$(x_0 \neq 0)$
	`  unsigned r = k % m;`	$\mathtt{r} \mapsto (y_0 \,\%\, x_0)$
	`  k = m; m = r;`	$\mathtt{k} \mapsto x_0, \mathtt{m} \mapsto (y_0 \,\%\, x_0)$
	`}`	
	`return k;`	

Automated Test-Case Generation

- E.g., want to cover `else`-branch at ①, loop at ② once

	`unsigned k, m;`	$\mathtt{x} \mapsto x_0, \mathtt{y} \mapsto y_0$
①	`if (x > y) {`	$(x_0 \leq y_0)$
	`  k = x; m = y`	
	`} else {`	
	`  k = y; m = x;`	$\mathtt{k} \mapsto y_0, \mathtt{m} \mapsto x_0$
	`}`	
②	`while (m != 0) {`	$(x_0 \neq 0)$
	`  unsigned r = k % m;`	$\mathtt{r} \mapsto (y_0 \,\%\, x_0)$
	`  k = m; m = r;`	$\mathtt{k} \mapsto x_0, \mathtt{m} \mapsto (y_0 \,\%\, x_0)$
	`}`	
	`return k;`	$((y_0 \,\%\, x_0) = 0)$

Automated Test-Case Generation

- We generated the constraint

$$(x_0 \leq y_0) \wedge (x_0 \neq 0) \wedge ((y_0 \,\%\, x_0) = 0)$$

- Is it *satisfiable*?

Automated Test-Case Generation

- We generated the constraint

$$(x_0 \leq y_0) \wedge (x_0 \neq 0) \wedge ((y_0 \,\%\, x_0) = 0)$$

- Is it *satisfiable*?
  - Yes, for instance $x_0 \mapsto 1, y_0 \mapsto 1$

Automated Test-Case Generation

- We generated the constraint

$$(x_0 \leq y_0) \land (x_0 \neq 0) \land ((y_0 \,\%\, x_0) = 0)$$

- Is it *satisfiable*?
  - Yes, for instance $x_0 \mapsto 1, y_0 \mapsto 1$
- Run *oracle* on input $x_0 \mapsto 1, y_0 \mapsto 1$

Automated Test-Case Generation

- We generated the constraint

$$(x_0 \leq y_0) \wedge (x_0 \neq 0) \wedge ((y_0 \,\%\, x_0) = 0)$$

- Is it *satisfiable*?
    - Yes, for instance $x_0 \mapsto 1, y_0 \mapsto 1$
- Run *oracle* on input $x_0 \mapsto 1, y_0 \mapsto 1$
    - We obtain the result 1

Automated Test-Case Generation

- We generated the constraint

$$(x_0 \leq y_0) \wedge (x_0 \neq 0) \wedge ((y_0 \,\%\, x_0) = 0)$$

- Is it *satisfiable*?
  - Yes, for instance $x_0 \mapsto 1, y_0 \mapsto 1$
- Run *oracle* on input $x_0 \mapsto 1, y_0 \mapsto 1$
  - We obtain the result 1
- Report test case, and select next path

Recall: Manual Test-Case Generation

```
unsigned gcd (unsigned x, unsigned y)
```

- Which equivalence classes would you generate?
- Which test cases would boundary testing yield?

If You Don’t Trust Testing . . .

. . . you can try to *prove the program correct.*

- An assertion is an (loop) invariant if
  - it holds upon loop entry
  - remains true after each iteration of the loop
- An invariant is *inductive*
  - if its validity upon loop entry is sufficient to guarantee that it still holds after the iteration

Euclid’s Algorithm and Inductive Invariants

Assume we have a *predicate GCD* with the following properties:

- $GCD(x, y) = GCD(y, x)$
- $GCD(0, x) = x$
- $GCD(x, x) = x$
- $(x > y) \Rightarrow GCD(x, y) = GCD(x\%y, y)$

```
while (m != 0) {

  unsigned r = k % m;

  k = m;

  m = r;

}
```

Euclid's Algorithm and Inductive Invariants

Assume we have a *predicate GCD* with the following properties:

- $GCD(x, y) = GCD(y, x)$
- $GCD(0, x) = x$
- $GCD(x, x) = x$
- $(x > y) \Rightarrow GCD(x, y) = GCD(x\%y, y)$

```
while (m != 0) {

  unsigned r = k % m;

  k = m;

  m = r;
  assert ($(k \geq m) \wedge GCD(x, y) = GCD(k, m)$);
}
```

Euclid's Algorithm and Inductive Invariants

Assume we have a *predicate GCD* with the following properties:

- $GCD(x, y) = GCD(y, x)$
- $GCD(0, x) = x$
- $GCD(x, x) = x$
- $(x > y) \Rightarrow GCD(x, y) = GCD(x\%y, y)$

```
while (m != 0) {

  unsigned r = k % m;

  k = m;
  assert ($(k \geq r) \wedge GCD(x, y) = GCD(k, r)$);
  m = r;
  assert ($(k \geq m) \wedge GCD(x, y) = GCD(k, m)$);
}
```

Euclid’s Algorithm and Inductive Invariants

Assume we have a *predicate GCD* with the following properties:

- $GCD(x, y) = GCD(y, x)$
- $GCD(0, x) = x$
- $GCD(x, x) = x$
- $(x > y) \Rightarrow GCD(x, y) = GCD(x\%y, y)$

```
while (m != 0) {

  unsigned r = k % m;
  assert ($(m \geq r) \land GCD(x, y) = GCD(m, r)$);
  k = m;
  assert ($(k \geq r) \land GCD(x, y) = GCD(k, r)$);
  m = r;
  assert ($(k \geq m) \land GCD(x, y) = GCD(k, m)$);
}
```

Euclid’s Algorithm and Inductive Invariants

Assume we have a *predicate GCD* with the following properties:

- $GCD(x, y) = GCD(y, x)$
- $GCD(0, x) = x$
- $GCD(x, x) = x$
- $(x > y) \Rightarrow GCD(x, y) = GCD(x\%y, y)$

```
while (m != 0) {
  assert ($(m \geq (k\%m)) \wedge GCD(x, y) = GCD(m, (k\%m))$);
  unsigned r = k % m;
  assert ($(m \geq r) \wedge GCD(x, y) = GCD(m, r)$);
  k = m;
  assert ($(k \geq r) \wedge GCD(x, y) = GCD(k, r)$);
  m = r;
  assert ($(k \geq m) \wedge GCD(x, y) = GCD(k, m)$);
}
```

Euclid's Algorithm and Inductive Invariants

Assume we have a *predicate GCD* with the following properties:

- $GCD(x, y) = GCD(y, x)$
- $GCD(0, x) = x$
- $GCD(x, x) = x$
- $(x > y) \Rightarrow GCD(x, y) = GCD(x\%y, y)$

```
while (m != 0) {
  assert ($(\underbrace{(m \geq (k\%m))}_{\texttt{true}} \land GCD(x, y) = GCD(m, (k\%m)))$);
  unsigned r = k % m;
  assert ($((m \geq r) \land GCD(x, y) = GCD(m, r))$);
  k = m;
  assert ($((k \geq r) \land GCD(x, y) = GCD(k, r))$);
  m = r;
  assert ($((k \geq m) \land GCD(x, y) = GCD(k, m))$);
}
```

Euclid’s Algorithm and Inductive Invariants

Assume we have a *predicate GCD* with the following properties:

- $GCD(x, y) = GCD(y, x)$
- $GCD(0, x) = x$
- $GCD(x, x) = x$
- $(x > y) \Rightarrow GCD(x, y) = GCD(x\%y, y)$

```
while (m != 0) {
  assert ($GCD(x, y) = GCD(m, (k\%m))$);
  ...
  assert ($(k \geq m) \wedge GCD(x, y) = GCD(k, m)$);
}
```

Euclid's Algorithm and Inductive Invariants

Assume we have a *predicate GCD* with the following properties:

- $GCD(x, y) = GCD(y, x)$
- $GCD(0, x) = x$
- $GCD(x, x) = x$
- $(x > y) \Rightarrow GCD(x, y) = GCD(x\%y, y)$

```
while (m != 0) {
  assert ($GCD(x, y) = GCD(m, (k\%m))$);
  ...
  assert ($(k \geq m) \wedge GCD(x, y) = GCD(k, m)$);
}
```

Need to show:

$$(k \geq m) \wedge (GCD(x, y) = GCD(k, m)) \Rightarrow (GCD(x, y) = GCD(m, (k\%m)))$$

Euclid’s Algorithm and Inductive Invariants

Assume we have a *predicate GCD* with the following properties:

- $GCD(x, y) = GCD(y, x)$
- $GCD(0, x) = x$
- $GCD(x, x) = x$
- $(x > y) \Rightarrow GCD(x, y) = GCD(x\%y, y)$

Need to show:

$$(k \geq m) \wedge (GCD(x, y) = GCD(k, m)) \Rightarrow (GCD(x, y) = GCD(m, (k\%m)))$$

Euclid's Algorithm and Inductive Invariants

Assume we have a *predicate GCD* with the following properties:

- $GCD(x, y) = GCD(y, x)$
- $GCD(0, x) = x$
- $GCD(x, x) = x$
- $(x > y) \Rightarrow GCD(x, y) = GCD(x\%y, y)$

Need to show:

$$(k \geq m) \wedge (GCD(x, y) = GCD(k, m)) \Rightarrow (GCD(x, y) = GCD(m, (k\%m)))$$

- Since $(k \geq m)$, we have $GCD(k, m) = GCD((k\%m), m)$

Euclid's Algorithm and Inductive Invariants

Assume we have a *predicate GCD* with the following properties:

- $GCD(x, y) = GCD(y, x)$
- $GCD(0, x) = x$
- $GCD(x, x) = x$
- $(x > y) \Rightarrow GCD(x, y) = GCD(x\%y, y)$

Need to show:

$$(k \geq m) \wedge (GCD(x, y) = GCD(k, m)) \Rightarrow (GCD(x, y) = GCD(m, (k\%m)))$$

- Since $(k \geq m)$, we have $GCD(k, m) = GCD((k\%m), m)$
- Therefore $GCD(x, y) = GCD(m, (k\%m))$

Euclid's Algorithm and Inductive Invariants

Assume we have a *predicate GCD* with the following properties:

- $GCD(x, y) = GCD(y, x)$
- $GCD(0, x) = x$
- $GCD(x, x) = x$
- $(x > y) \Rightarrow GCD(x, y) = GCD(x\%y, y)$

Need to show:

$$(k \geq m) \wedge (GCD(x, y) = GCD(k, m)) \Rightarrow (GCD(x, y) = GCD(m, (k\%m)))$$

- Since $(k \geq m)$, we have $GCD(k, m) = GCD((k\%m), m)$
- Therefore $GCD(x, y) = GCD(m, (k\%m))$
- Loop iteration does not invalidate

$$(k \geq m) \wedge GCD(x, y) = GCD(k, m)$$

Euclid’s Algorithm and Inductive Invariants

Does

$$(k \geq m) \wedge GCD(x, y) = GCD(k, m)$$

hold at the beginning of the loop?

```
unsigned k = max(x,y);
unsigned m = min(x,y);
```

Euclid's Algorithm and Inductive Invariants

Does

$$(k \geq m) \wedge GCD(x, y) = GCD(k, m)$$

guarantee that $k = GCD(x, y)$ after the loop?

- After the loop, we know that $m = 0$
- Therefore

$$(k \geq 0) \wedge GCD(x, y) = GCD(k, 0)$$

Euclid’s Algorithm and Inductive Invariants

Does

$$(k \geq m) \wedge GCD(x, y) = GCD(k, m)$$

guarantee that $k = GCD(x, y)$ after the loop?

- After the loop, we know that $m = 0$
- Therefore

$$(k \geq 0) \wedge GCD(x, y) = GCD(k, 0)$$

- The algorithm is correct!

Symbolic Simulation with KLEE

`http://klee.github.io`

- Explores paths of LLVM programs
- Symbolic simulation for test-case generation

Symbolic Simulation with Klee

```c
#include <klee/klee.h>

int get_sign(int x) {
  if (x == 0)
    return 0;
  if (x<0)
    return -1;
  else
    return 1;
}

int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

KLEE Tutorial (Docker Image)

Try at home:

- Docker (`https://www.docker.com/get-docker`)
- Instructions on

  ```
  klee.github.io/tutorials/
  ```

- Load/Create Docker Image:

  ```
  docker run -ti --name=klee_psv
  --ulimit='stack=-1:-1' klee/klee
  ```

- Restart (after `exit`):

  ```
  docker start -ai klee_psv
  ```

KLEE Tutorial (http://klee.github.io/tutorials/)

Trivial example from before (get_sign):

- In the get_sign directory:

```
cd /home/klee/klee_src/examples/get_sign
```

- Translate source to LLVM bitcode:

```
clang -I ../../include -emit-llvm -c -g
                get_sign.c
```

- Run KLEE on the generated bitcode:

```
klee get_sign.bc
```

KLEE Tutorial (`http://klee.github.io/tutorials/`)

- KLEE generates several test-cases in `klee-out-0`
- Inputs can be viewed using the following command:

```
ktest-tool test000001.ktest
```

- Replay test-cases:
  - `clang -I ../../include/ -L /home/klee/klee_build/lib/ get_sign.c -lkleeRuntest`
  - `export LD_LIBRARY_PATH=/home/klee/klee_build/lib/`
  - `KTEST_FILE=klee-last/test000001.ktest ./a.out`
  - `echo $?`

Checking Coverage with `clang` and `llvm-cov`

Compile with coverage instrumentation:

- `clang --coverage -I ../../include/ -L /home/klee/klee_build/lib/ get_sign.c -lkleeRuntest`

Run tests as before:

- `KTEST_FILE=klee-last/test000001.ktest ./a.out`

Show coverage information:

- `llvm-cov gcov get_sign.gcno`

Try this with gcd!

```
#include <klee/klee.h>
#define MAX(x, y) ((x)<(y))?(y):(x)
unsigned gcd (unsigned x, unsigned y)
{
  unsigned k = x;
  unsigned m = y;
  if ((x==0) || (y==0)) return MAX(x, y);
  while (k != m) {
    if (k > m) k = k - m;
    else m = m - k;
  }
  return k;
}

int main(int argc, char** argv)
{
  unsigned a, b;
  klee_make_symbolic (&a, sizeof(a), "a");
  klee_make_symbolic (&b, sizeof(b), "b");
  return gcd (a, b);
}
```

Try this with gcd!

```c
#include <klee/klee.h>
#define MIN(x, y) ((x)<(y))?(x):(y)
#define MAX(x, y) ((x)<(y))?(y):(x)

unsigned gcd (unsigned x, unsigned y)
{
  unsigned k = MAX (x,y);
  unsigned m = MIN (x,y);
  while (m != 0) {
    unsigned r = k % m;
    k = m; m = r;
  }
  return k;
}
int main(int argc, char** argv)
{
  unsigned a, b;
  klee_make_symbolic (&a, sizeof(a), "a");
  klee_make_symbolic (&b, sizeof(b), "b");
  return gcd (a, b);
}
```

Exploring `gcd` with KLEE

- On `gcd`, KLEE doesn't terminate! (Why?)

Exploring `gcd` with KLEE

- On `gcd`, KLEE doesn't terminate! (Why?)
- Restrict run-time:
    - `-max-time=`*n* (halt after *n* seconds)
    - `-max-fork=`*n* (stop forking after *n* symbolic branches)
    - `-max-memory=`*n* (limit memory consumption to *n* megabytes)
    - or simply use Ctrl+C…

Exploring `gcd` with KLEE

- On `gcd`, KLEE doesn't terminate! (Why?)
- Restrict run-time:
  - `-max-time=`*n* (halt after *n* seconds)
  - `-max-fork=`*n* (stop forking after *n* symbolic branches)
  - `-max-memory=`*n* (limit memory consumption to *n* megabytes)
  - or simply use Ctrl+C...
- We can apply test-cases generated for prototype to `gcd`!
  - Simply make sure that the symbolic variables are the same!

Hints for Using Docker

- Copy a file from host to Docker image:

```
docker cp gcd.c klee_psv:/home/klee/gcd.c
```

- “Got permission denied while trying to connect . . . ” error:

```
usermod -a -G docker $USER
```

  (or run using `sudo` if that fails)

Symbolic Simulation with KLEE

- Also supports complex build systems (WLLVM)
- Can be used as LLVM-bitcode interpreter
    - Check `coreutils` tutorial on KLEE webpage
- Supports symbolic command-line parameters
    - using a dedicated library; check `http://klee.github.io/tutorials/testing-coreutils/`

Assertions

Byte swapping trick:

- `assert(x==y); x=x^y; y=x^y; x=x^y; assert(x==y);`

```
x=x^y;

y=x^y;

x=x^y;
assert(x==y);
```

Assertions

Byte swapping trick:

- `assert(x==y); x=x^y; y=x^y; x=x^y; assert(x==y);`

```
x=x^y;

y=x^y;
assert((x^y)==y);
x=x^y;
assert(x==y);
```

Assertions

Byte swapping trick:

- `assert(x==y); x=x^y; y=x^y; x=x^y; assert(x==y);`

```
x=x^y;
assert((x^(x^y))==(x^y));
y=x^y;
assert((x^y)==y);
x=x^y;
assert(x==y);
```

Assertions

Byte swapping trick:

- `assert(x==y); x=x^y; y=x^y; x=x^y; assert(x==y);`

```
assert(((x^y)^((x^y)^y))==((x^y)^y));
x=x^y;
assert((x^(x^y))==(x^y));
y=x^y;
assert((x^y)==y);
x=x^y;
assert(x==y);
```

Assertions

Byte swapping trick:

- `assert(x==y); x=x^y; y=x^y; x=x^y; assert(x==y);`

```
assert(((x^y)^((x^y)^y))==((x^y)^y));
x=x^y;
assert((x^(x^y))==(x^y));
y=x^y;
assert((x^y)==y);
x=x^y;
assert(x==y);
```

- We know that `x^y = y^x`

$$\underbrace{\text{(x\^{}y)\^{}((x\^{}y)\^{}y)}}_{\text{x\^{}x\^{}y\^{}y\^{}y}} = \text{(x\^{}y)\^{}y}$$

Assertions

Byte swapping trick:

- `assert(x==y); x=x^y; y=x^y; x=x^y; assert(x==y);`

```
assert(((x^y)^((x^y)^y))==((x^y)^y));
x=x^y;
assert((x^(x^y))==(x^y));
y=x^y;
assert((x^y)==y);
x=x^y;
assert(x==y);
```

- We know that `x^y` = `y^x`

$$\underbrace{(x\hat{}y)\hat{}((x\hat{}y)\hat{}y)}_{x\hat{}x\hat{}y\hat{}y\hat{}y} = (x\hat{}y)\hat{}y$$

- Furthermore `x^x` = 0 and `x^0` = `x`, therefore we obtain $(y = x)$

Assertions and Concurrency

- *Locks* can be used to prevent simultaneous or concurrent access to critical regions or resources
- Simplified API:
    - `lock(A)` succeeds if lock `A` is available
    - `lock(A)` blocks if lock is already held/acquired (by this or another thread)
    - `unlock(A)` releases a lock previously acquired
    - `unlock(A)` never blocks

Assertions and Concurrency

- Deadlocks happen if locks are acquired in wrong order

```
lock (A);
  lock (B);
  unlock (B);
unlock (A);
```

```
lock (B);
  lock (A);
  unlock (A);
unlock (B);
```

Assertions and Concurrency

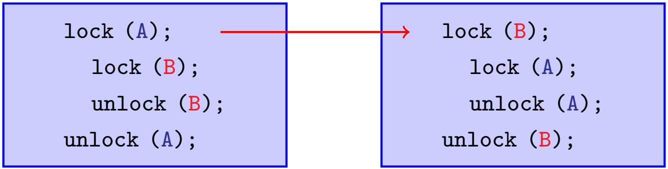
- Deadlocks happen if locks are acquired in wrong order
  - Thread one acquires lock A

```
lock (A);
  lock (B);
  unlock (B);
unlock (A);
```

```
lock (B);
  lock (A);
  unlock (A);
unlock (B);
```

Assertions and Concurrency

- Deadlocks happen if locks are acquired in wrong order
  - Thread one acquires lock A
  - Thread two acquires lock B

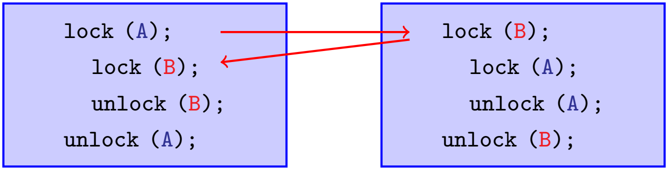


```
lock (A);
  lock (B);
  unlock (B);
unlock (A);
```

```
lock (B);
  lock (A);
  unlock (A);
unlock (B);
```

Assertions and Concurrency

- Deadlocks happen if locks are acquired in wrong order
  - Thread one acquires lock A
  - Thread two acquires lock B
  - Thread one waits for lock B (thread two still running)

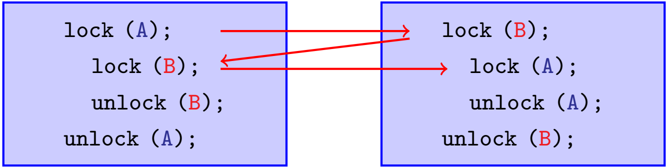


```
lock (A);
  lock (B);
  unlock (B);
unlock (A);
```

```
lock (B);
  lock (A);
  unlock (A);
unlock (B);
```

Assertions and Concurrency

- Deadlocks happen if locks are acquired in wrong order
  - Thread one acquires lock A
  - Thread two acquires lock B
  - Thread one waits for lock B
  - Thread two waits for lock A

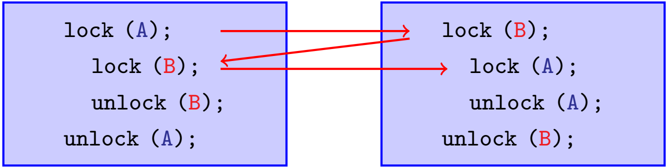


```
lock (A);
  lock (B);
  unlock (B);
unlock (A);
```

```
lock (B);
  lock (A);
  unlock (A);
unlock (B);
```

Assertions and Concurrency

- Deadlocks happen if locks are acquired in wrong order
  - Thread one acquires lock A
  - Thread two acquires lock B
  - Thread one waits for lock B
  - Thread two waits for lock A
  - Now both threads are stuck. . .



```
lock (A);
  lock (B);
  unlock (B);
unlock (A);
```

```
lock (B);
  lock (A);
  unlock (A);
unlock (B);
```

The Task

- Add assertions that fail if a deadlock is about to occur!
- Assertions must *not* fail if no deadlock occurs!
- **Hints:**
  - You need to augment the code with auxiliary code and variables indicating when a process is waiting for a lock
  - The assertions must be executed *before* the deadlock occurs

For the specialists among you: assume sequential consistency

Solution for Deadlocks

```
flagA = 0;
lock (A);
  flagA = 1;
  assert (!flagB);
  lock (B);
  flagA = 0;
  unlock (B);
unlock (A);
```

```
flagB = 0;
lock (B);
  flagB = 1;
  assert (!flagA);
  lock (A);
  flagB = 0;
  unlock (A);
unlock (B);
```

Note:

- If only one thread contains an assertion, then there's a potential deadlock without an assertion failure
- If `flagA` and `flagB` are reset after the inner locks are released, then there's a potential assertion failure even if the deadlock doesn't happen

Inductive Invariants

- Add an *inductive invariant* to the code
- Use it to show that the assertion after the loop holds
- Add comments to the code explaining
  - why your assertion is an inductive invariant
  - why it shows that the assertion after the loop holds

```
unsigned x = i;
unsigned y = j;
while (x != 0)
{
  x--;
  y++;
  assert (?); // add invariant here
}
assert ((i != j) || (y == 2 * i));
```

Solution for Invariant

```
assert (j == j + (i - i));
int x = i;
assert (j == j + (i - x));
int y = j;
assert (y == j + (i - x));
while (x != 0) {
  assert ((y + 1) == j + (i - (x - 1)));
  x--;
  assert ((y + 1) == j + (i - x));
  y++;
  assert (y == j + (i - x)); // # iterations n := i - x
}
assert ((x == 0) && y == j + (i - x));
assert ((i != j) || (y == 2 * i));
```

Solution for Invariant (ctd.)

- `(y==j+(i-x))` implies `(y+1)==j+(i-(x-1))`
  - Therefore `(y==j+(i-x))` is a loop invariant
- `(y==j+(i-x))` is inductive
  - Holds at beginning of loop, since `(j == j + (i - i))` is true
- Implies assertion after loop (since `x == 0`)

Summary

Today was a recap of

- Assertions
- Testing
- Test Case Generation
- Inductive Invariants

Summary

Today was a recap of

- Assertions
- Testing
- Test Case Generation
- Inductive Invariants

Next time it's getting a bit more *formal*