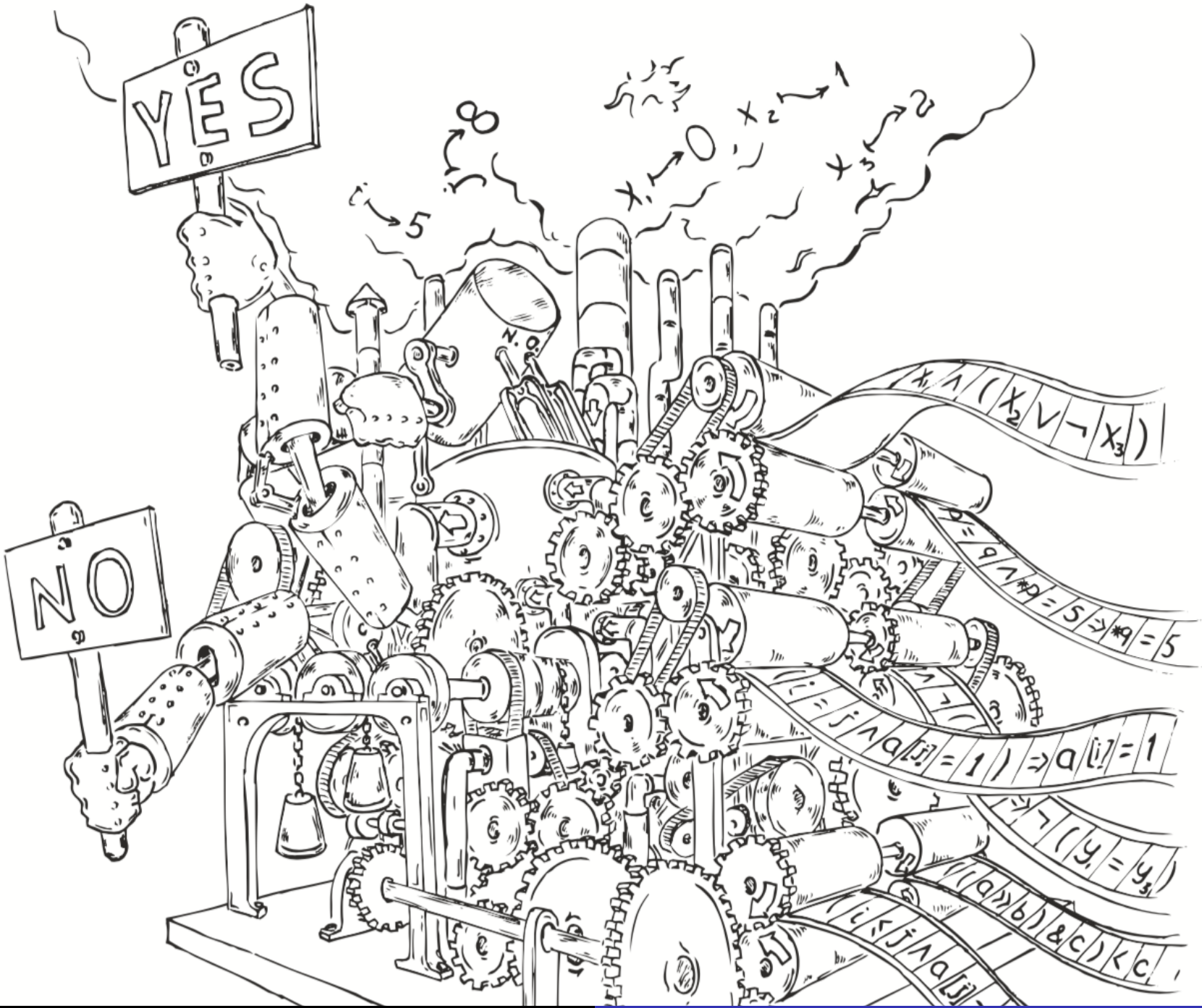

YES
NO

SAT/SMT solvers
10. Program Analysis

Roman Kholin

Lalambda

Tbilisi, 2023

Why

- automatic test generation
- detection of dead code
- verification of properties given in the form of assertions

Example

```
1  void ReadBlocks(int data[], int cookie)
2  {
3    int i = 0;
4    while (true)
5    {
6      int next;
7      next = data[i];
8      if (!(i < next && next < N)) return;
9      i = i + 1;
10     for (; i < next; i = i + 1) {
11         if (data[i] == cookie)
12             i = i + 1;
13         else
14             Process(data[i]);
15     }
16   }
17 }
```

Example

Line	Kind	Instruction or condition
3	Assignment	`i = 0;`
7	Assignment	`next = data[i];`
8	Branch	`i < next && next < N`
9	Assignment	`i = i + 1;`
10	Branch	`i < next`
11	Branch	`data[i] != cookie`
14	Function call	`Process(data[i]);`
10	Assignment	`i = i + 1;`
10	Branch	`!(i < next)`
7	Assignment	`next = data[i];`
8	Branch	`!(i < next && next < N)`

Static single assignment

Line	Kind	Instruction or condition
3	Assignment	$i_1 = 0;$
7	Assignment	$next_1 = data_0[i_1];$
8	Branch	$i_1 < next_1 \;\&\&\; next_1 < N_0$
9	Assignment	$i_2 = i_1 + 1;$
10	Branch	$i_2 < next_1$
11	Branch	$data_0[i_2] \;!=\; cookie_0$
14	Function call	$Process(data_0[i_2]);$
10	Assignment	$i_3 = i_2 + 1;$
10	Branch	$!(i_3 < next_1)$
7	Assignment	$next_2 = data_0[i_3];$
8	Branch	$!(i_3 < next_2 \;\&\&\; next_2 < N_0)$

Path constraint

$$
\begin{aligned}
& i_1 = 0 && \wedge \\
& next_1 = data_0[i_1] && \wedge \\
& (i_1 < next_1 \ \wedge \ next_1 < N_0) && \wedge \\
& i_2 = i_1 + 1 && \wedge \\
& i_2 < next_1 && \wedge \\
& data_0[i_2] = cookie_0 && \wedge \\
& i_3 = i_2 + 1 && \wedge \\
& i_4 = i_3 + 1 && \wedge \\
& \neg(i_4 < next_1) && \wedge \\
& \neg(0 \leq i_4 \ \wedge \ i_4 < N_0)
\end{aligned}
$$

Example

```
1  void ReadBlocks(int data[], int cookie)
2  {
3    int i = 0;
4    while (true)
5    {
6      int next;
7      next = data[i];
8      if (!(i < next && next < N)) return;
9      i = i + 1;
10     for (; i < next; i = i + 1) {
11         if (data[i] == cookie)
12             i = i + 1;
13         else
14             Process(data[i]);
15     }
16   }
17 }
```

Error

$$i_1 = 0 \wedge \neg(0 \leq i_1 \wedge i_1 < N_0)$$

Error

$$i_1 = 0 \wedge \neg(0 \leq i_1 \wedge i_1 < N_0)$$
$$\{i_1 := 0, N_0 := 0\}$$

Another trace

Line	Kind	Instruction or condition
3	Assignment	`i = 0;`
7	Assignment	`next = data[i];`
8	Branch	`i < next && next < N`
9	Assignment	`i = i + 1;`
10	Branch	`i < next`
11	Branch	`data[i] = cookie`
12	Assignment	`i = i + 1;`
10	Assignment	`i = i + 1;`
10	Branch	`!(i < next)`
7	Assertion	`0 <= i && i < N`

Another trace

$$
\begin{aligned}
& i_1 = 0 && \wedge \\
& next_1 = data_0[i_1] && \wedge \\
& (i_1 < next_1 \ \wedge \ next_1 < N_0) && \wedge \\
& i_2 = i_1 + 1 && \wedge \\
& i_2 < next_1 && \wedge \\
& data_0[i_2] = cookie_0 && \wedge \\
& i_3 = i_2 + 1 && \wedge \\
& i_4 = i_3 + 1 && \wedge \\
& \neg(i_4 < next_1) && \wedge \\
& \neg(0 \leq i_4 \ \wedge \ i_4 < N_0)
\end{aligned}
$$

Assigment of input

$$\{i_1 \mapsto 0,\ N_0 \mapsto 3,\ next_1 \mapsto 2,\ data_0 \mapsto \langle 2, 6, 5\rangle,$$
$$i_2 \mapsto 1,\ cookie_0 \mapsto 6,\ i_3 \mapsto 2,\ i_4 \mapsto 3\}\ ,$$

Bounded Program Analysis

```
1  if (i < next) {
2    if (data[i] == cookie)
3       i = i + 1;
4    else
5       Process(data[i]);
6
7    i = i + 1;
8
9    if (i < next) {
10     if (data[i] == cookie)
11        i = i + 1;
12     else
13        Process(data[i]);
14
15     i = i + 1;
16   }
17 }
```

```
1  γ1 = (i0 < next0);
2  γ2 = (data0[i1] == cookie0);
3  i1 = i0 + 1;
4
5
6  i2 = γ2 ? i1 : i0;
7  i3 = i2 + 1;
8
9  γ3 = (i3 < next0);
10 γ4 = (data0[i3] == cookie0);
11 i4 = i3 + 1;
12
13
14 i5 = γ4 ? i4 : i3;
15 i6 = i5 + 1;
16 i7 = γ3 ? i6 : i3;
17 i8 = γ1 ? i7 : i0;
```

Unbounded Program Analysis

```
1 int i=0, j=0;
2
3 while(data[i] != '\n')
4 {
5   i++;
6   j=i;
7 }
8
9 assert(i == j);
```

⟶

```
1 int i=0, j=0;
2
3 if(data[i] != '\n')
4 {
5   i=*;
6   j=*;
7   i++;
8   j=i;
9 }
10
11 assume(data[i] == '\n')
12
13 assert(i == j);
```

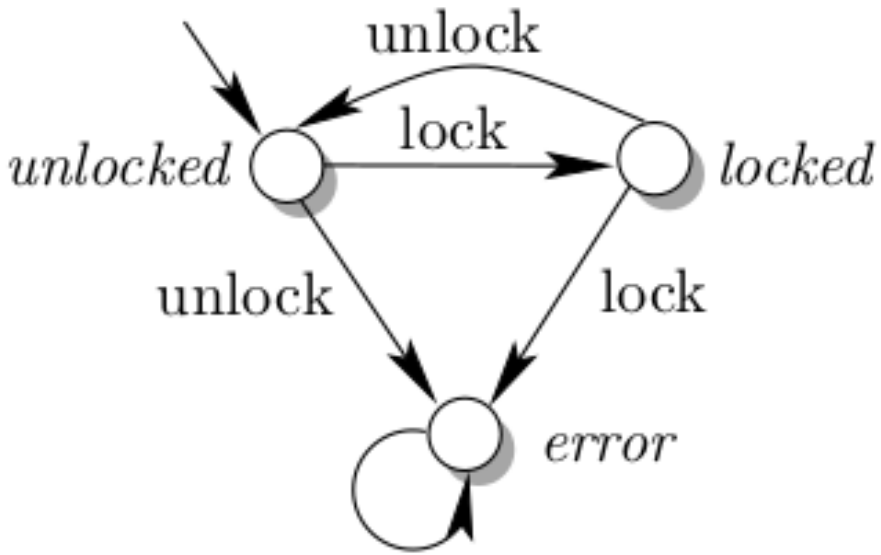
Path constraint

$$
\begin{aligned}
& i_1 = 0 && \wedge \\
& j_1 = 0 && \wedge \\
& \gamma_1 = (data_0[i_1] \neq \texttt{'\textbackslash n'}) && \wedge \\
& i_3 = i_2 + 1 && \wedge \\
& j_3 = i_3 && \wedge \\
& i_4 = \gamma_1 \,?\, i_3 : i_1 && \wedge \\
& j_4 = \gamma_1 \,?\, j_3 : j_1 && \wedge \\
& data_0[i_4] = \texttt{'\textbackslash n'} && \wedge \\
& i_4 \neq j_4
\end{aligned}
$$

Bad approximation

```
1    do {
2      lock();
3      old_count = count;
4      request = GetNextRequest();
5      if (request != NULL) {
6        ReleaseRequest(request);
7        unlock();
8        ProcessRequest(request);
9        count = count + 1;
10     }
11   }
12   while(old_count != count);
13   unlock();
```

Automat of lock



Bad approximation

```
1 int i=0;
2
3 while(i != 10) {
4       ...
5       i++;
6 }
```

Asserts

```
1    state_of_lock = unlocked;
2    do {
3         assert(state_of_lock == unlocked);
4         state_of_lock = locked;
5         old_count = count;
6         request = GetNextRequest();
7         if (request != NULL) {
8            ReleaseRequest(request);
9            assert(state_of_lock == locked);
10           state_of_lock = unlocked;
11           ProcessRequest(request);
12           count = count + 1;
13        }
14   }
15   while (old_count != count);
16   assert(state_of_lock == locked);
17   state_of_lock = unlocked;
```

Invariants

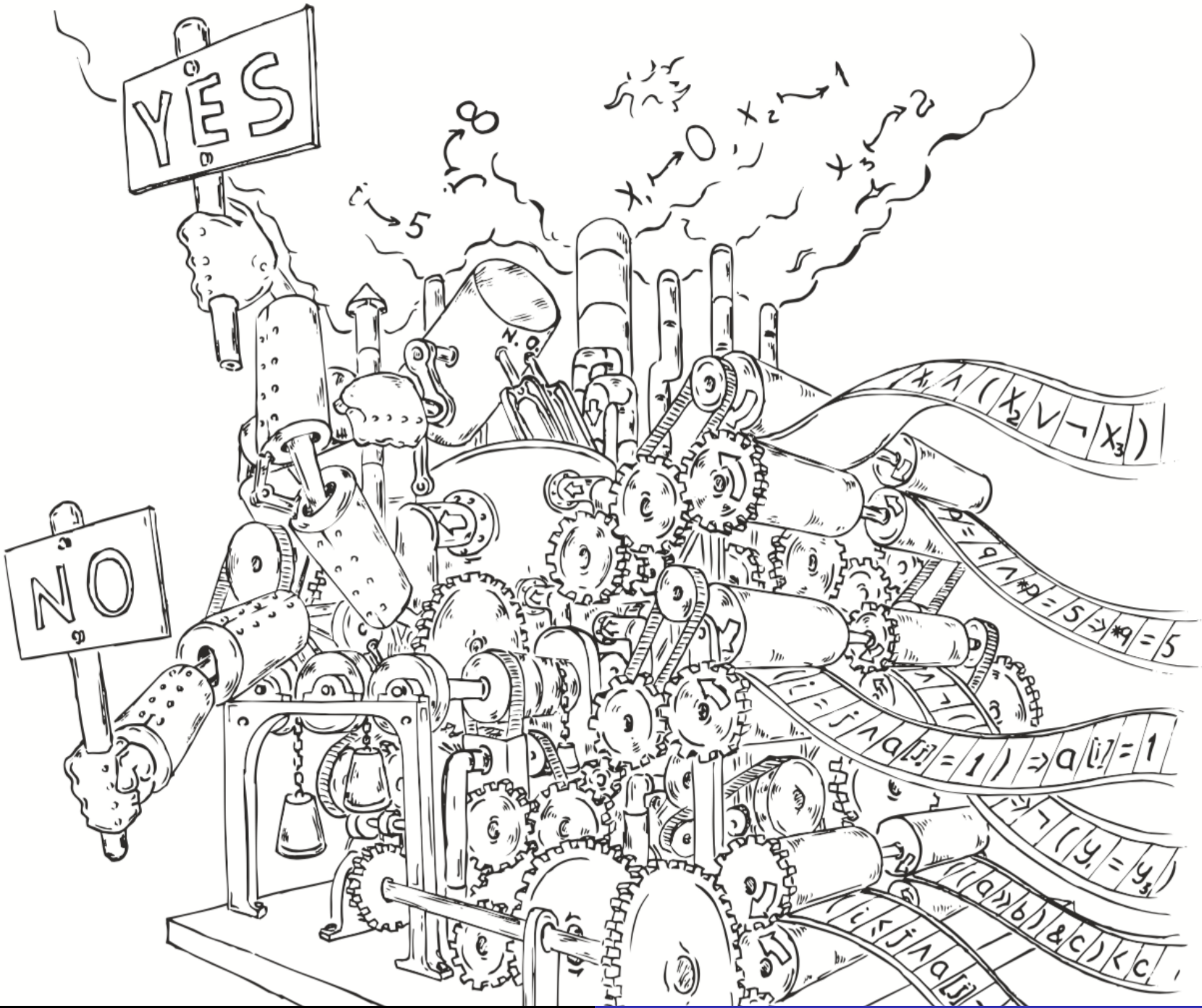
```
1 int i=0;
2
3 while(i != 10) {
4       ...
5       i++;
6 }
```

Invariants

```
1 A;
2 while(C) {
3       assert(I);
4       B;
5 }
```

```
1    state_of_lock = unlocked;
2
3    state_of_lock = *;
4    old_count = *;
5    count = *;
6    request = *;
7
8    assert(state_of_lock == unlocked);
9    state_of_lock = locked;
10   old_count = count;
11   request = GetNextRequest();
12   if (request != NULL) {
13     ReleaseRequest(request);
14     assert(state_of_lock == locked);
15     state_of_lock = unlocked;
16     ProcessRequest(request);
17     count = count + 1;
18   }
19
20   assume(old_count == count);
21
22   assert(state_of_lock == locked);
23   state_of_lock = unlocked;
```

```
1   state_of_lock = unlocked;
2
3   assert(state_of_lock == unlocked); // induction base case
4
5   state_of_lock = *;
6   old_count = *;
7   count = *;
8   request = *;
9
10  assume(state_of_lock == unlocked); // induction hypothesis
11
12  assert(state_of_lock == unlocked); // property
13  state_of_lock = locked;
14  old_count = count;
15  request = GetNextRequest();
16  if (request != NULL) {
17    ReleaseRequest(request);
18    assert(state_of_lock == locked); // property
19    state_of_lock = unlocked;
20    ProcessRequest(request);
21    count = count + 1;
22  }
23
24   // induction step case
25  assert(old_count != count ⟹ state_of_lock == unlocked);
26
27  assume(old_count == count);
28
29  assert(state_of_lock == locked); // property
30  state_of_lock = unlocked;
```


YES
NO