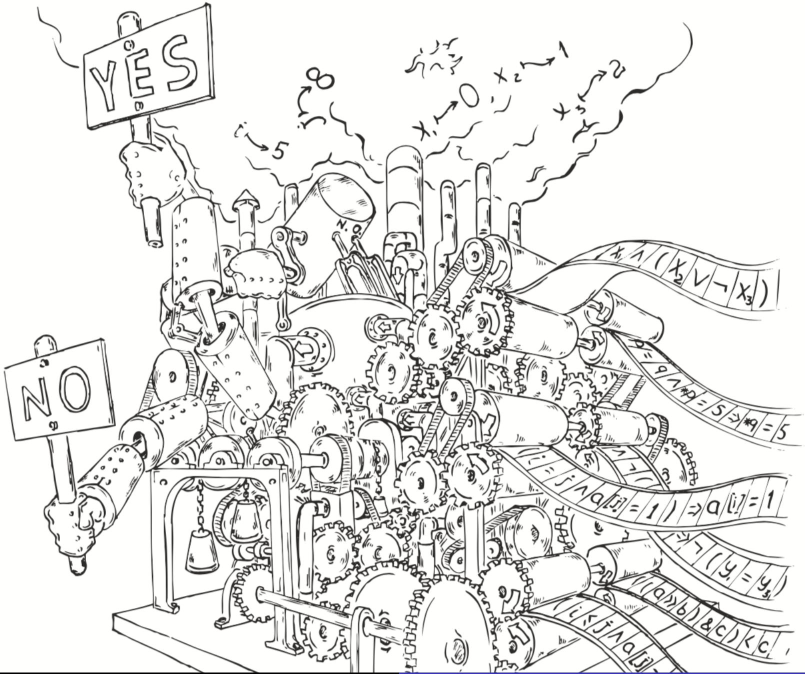

YES
NO
N.O.

SAT/SMT solvers
10. Program Analysis

Roman Kholin

Lomonosov Moscow State University

Moscow, 2023

Why

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

Example

```
void ReadBlocks(int data[], int cookie)
{
  int i = 0;
  while (true)
  {
    int next;
    next = data[i];
    if (!(i < next && next < N)) return;
    i = i + 1;
    for (; i < next; i = i + 1) {
        if (data[i] == cookie)
           i = i + 1;
        else
           Process(data[i]);
    }
  }
}
```

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
$\boxed{3}$	Assignment	`i`$_1$ `= 0;`
$\boxed{7}$	Assignment	`next`$_1$ `= data`$_0$`[i`$_1$`];`
$\boxed{8}$	Branch	`i`$_1$ `< next`$_1$ `&& next`$_1$ `<` N_0
$\boxed{9}$	Assignment	`i`$_2$ `= i`$_1$ `+ 1;`
$\boxed{10}$	Branch	`i`$_2$ `< next`$_1$
$\boxed{11}$	Branch	`data`$_0$ `[i`$_2$ `] != cookie`$_0$
$\boxed{14}$	Function call	`Process(data`$_0$ `[i`$_2$ `]);`
$\boxed{10}$	Assignment	`i`$_3$ `= i`$_2$ `+ 1;`
$\boxed{10}$	Branch	`!(i`$_3$ `< next`$_1$ `)`
$\boxed{7}$	Assignment	`next`$_2$ `= data`$_0$ `[i`$_3$ `];`
$\boxed{8}$	Branch	`!(i`$_3$ `< next`$_2$ `&& next`$_2$ `<` N_0`)`

Path constraint

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

Example

```
void ReadBlocks(int data[], int cookie)
{
  int i = 0;
  while (true)
  {
    int next;
    next = data[i];
    if (!(i < next && next < N)) return;
    i = i + 1;
    for (; i < next; i = i + 1) {
        if (data[i] == cookie)
           i = i + 1;
        else
           Process(data[i]);
    }
  }
}
```

Error

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

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

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

Assigment of input

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

```
if (i < next) {
  if (data[i] == cookie)
     i = i + 1;
  else
     Process(data[i]);

  i = i + 1;

  if (i < next) {
    if (data[i] == cookie)
       i = i + 1;
    else
       Process(data[i]);

    i = i + 1;
  }
}
```

$\gamma_1 = (\mathtt{i}_0 < \mathtt{next}_0);$

$\gamma_2 = (\mathtt{data}_0[\mathtt{i}_1] == \mathtt{cookie}_0);$

$\mathtt{i}_1 = \mathtt{i}_0 + 1;$

$\mathtt{i}_2 = \gamma_2 \ ?\ \mathtt{i}_1 : \mathtt{i}_0;$

$\mathtt{i}_3 = \mathtt{i}_2 + 1;$

$\gamma_3 = (\mathtt{i}_3 < \mathtt{next}_0);$

$\gamma_4 = (\mathtt{data}_0[\mathtt{i}_3] == \mathtt{cookie}_0);$

$\mathtt{i}_4 = \mathtt{i}_3 + 1;$

$\mathtt{i}_5 = \gamma_4 \ ?\ \mathtt{i}_4 : \mathtt{i}_3;$

$\mathtt{i}_6 = \mathtt{i}_5 + 1;$

$\mathtt{i}_7 = \gamma_3 \ ?\ \mathtt{i}_6 : \mathtt{i}_3;$

$\mathtt{i}_8 = \gamma_1 \ ?\ \mathtt{i}_7 : \mathtt{i}_0;$

Unbounded Program Analysis

```
int i=0, j=0;

while(data[i] != '\n')
{
  i++;
  j=i;
}

assert(i == j);
```

→

```
int i=0, j=0;

if(data[i] != '\n')
{
  i=*;
  j=*;
  i++;
  j=i;
}

assume(data[i] == '\n')

assert(i == j);
```

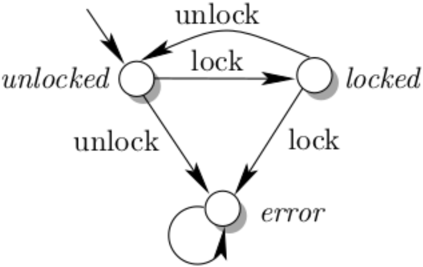
Path constraint

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

Bad approximation

```
do {
  lock();
  old_count = count;
  request = GetNextRequest();
  if (request != NULL) {
    ReleaseRequest(request);
    unlock();
    ProcessRequest(request);
    count = count + 1;
  }
}
while(old_count != count);
unlock();
```

Automat of lock



Asserts

```
state_of_lock = unlocked;
do {
    assert(state_of_lock == unlocked);
    state_of_lock = locked;
    old_count = count;
    request = GetNextRequest();
    if (request != NULL) {
        ReleaseRequest(request);
        assert(state_of_lock == locked);
        state_of_lock = unlocked;
        ProcessRequest(request);
        count = count + 1;
    }
}
while (old_count != count);
assert(state_of_lock == locked);
state_of_lock = unlocked;
```

Invariants

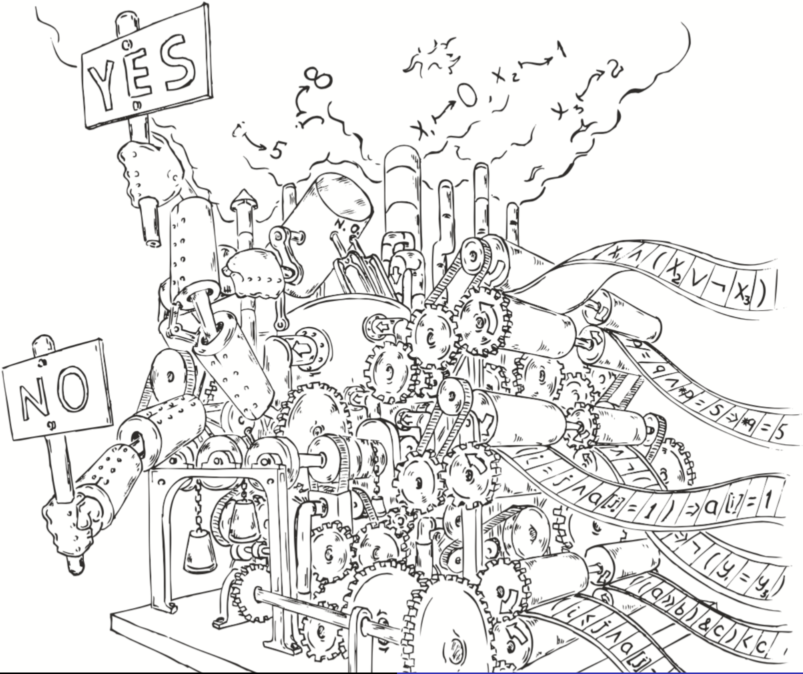
```
int i=0;

while(i != 10) {
      ...
      i++;
}
```

Invariants

```
A;
while(C) {
        assert(I);
        B;
}
```

```
state_of_lock = unlocked;

state_of_lock = *;
old_count = *;
count = *;
request = *;

assert(state_of_lock == unlocked);
state_of_lock = locked;
old_count = count;
request = GetNextRequest();
if (request != NULL) {
  ReleaseRequest(request);
  assert(state_of_lock == locked);
  state_of_lock = unlocked;
  ProcessRequest(request);
  count = count + 1;
}

assume(old_count == count);

assert(state_of_lock == locked);
state_of_lock = unlocked;
```

```
state_of_lock = unlocked;

assert(state_of_lock == unlocked); // induction base case

state_of_lock = *;
old_count = *;
count = *;
request = *;

assume(state_of_lock == unlocked); // induction hypothesis

assert(state_of_lock == unlocked); // property
state_of_lock = locked;
old_count = count;
request = GetNextRequest();
if (request != NULL) {
  ReleaseRequest(request);
  assert(state_of_lock == locked); // property
  state_of_lock = unlocked;
  ProcessRequest(request);
  count = count + 1;
}

 // induction step case
assert(old_count != count ⟹ state_of_lock == unlocked);

assume(old_count == count);

assert(state_of_lock == locked); // property
state_of_lock = unlocked;
```


YES
NO
N.O.