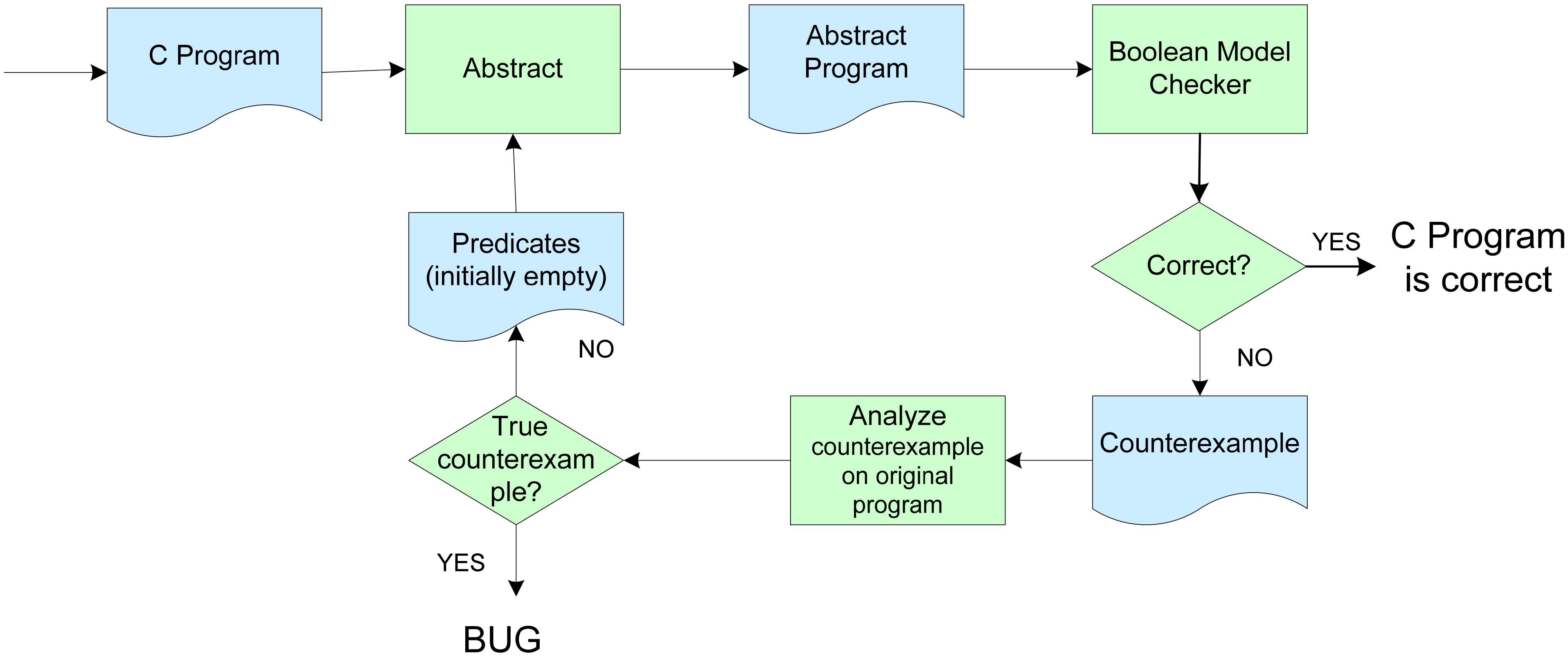
SLAM
Abstraction

Roderick Bloem

The Approach



Abstraction

- Represent complex program by simple program
  - original program is **concrete**, simple one is **abstract**
- Construction: if abstraction correct, then original correct
  - But: abstract program may fail even if the original is correct
  - We will look at *refinement* later
- Whenever we can not make a decision with certainty, we allow all possibilities

Predicate Abstraction

- Replace variables by predicates. E.g., instead of `x` have the predicates
  - `b, meaning {x>0},`
  - `c: {x<0},`
  - `d: {x==0}`
- or replace `x` and `y` by
  - `e: {x==y}, or by`
  - `f: {x<y}, or by`
  - `g: {2x - y < 0},`

Predicate Abstraction

Example: keep only the lowest bit of a number.

- `b: {x is odd}`
- `assert(x!=38)` becomes `assert(b)`
- `assert(b)` is stricter:
  - if `assert(x!=38)` fails then `assert(b)` fails
  - But not vice-versa

- `if(x==5) then S1 else S2 fi` becomes
  `if(b?*:F) then S1 else S2 fi`

(meaning: if b is true, try both branches, otherwise try only the else branch)

Construct abstract programs **one statement at a time**

Abstraction Example

For automatic abstraction, let’s first check some basics.

Let’s say we have one predicate:

`b = {x ≤ y}`

How do we abstract

```
x := y?
y := y+1?
```

Computing Abstraction

$b = \{x \leq y\}$

Use Hoare's weakest precondition

`x := y`

`x := y` is abstracted to

Computing Abstraction

$b = \{x \le y\}$

Use Hoare's weakest precondition

$\{y \le y\}$

$x := y$

$\{x \le y\}$

Thus, $y \le y$ before the statement iff $x \le y$ after

$x := y$ is abstracted to $b = \text{true}$

Computing Abstraction

Now for `y := y + 1`.
`b = {x ≤ y}`

Thus, $x \leq y + 1$ before iff $x \leq y$ after.
In which cases can we guarantee $x \leq y+1$?

b	b'
{x ≤ y}	{x ≤ y+1}
T	
F	

abstraction:

Computing Abstraction

Now for `y := y + 1`.

```
{x ≤ y + 1}
y := y + 1
{x ≤ y}
```

Thus, `x ≤ y + 1` before iff `x ≤ y` after.
In which cases can we guarantee `x ≤ y+1`?

b	b'
{x ≤ y}	{x ≤ y+1}
T	T
F	*

We don't have enough information to decide whether `x≤y+1` before, so we approximate.

abstraction: `b = b ? T : *;`

Conservative Abstraction

Let us abstract `x` by **b**:{**x** < 0}.

We may loose some information Example:

```
x = -2;
x = x + 1;
assert(x<0);
```

is abstracted statement-by statement-to

The abstraction is *conservative*: bugs are preserved (but new bugs may occur).

Conservative Abstraction

Let us abstract `x` by **b**:**{x < 0}.**

We may loose some information Example:

```
x = -2;
x = x + 1;
assert(x<0);
```

is abstracted statement-by statement-to

```
b = true;
b = b ? * : false;
assert(b);
```

The abstraction is *conservative*: bugs are preserved (but new bugs may occur).

Multiple Predicates

Two predicates: `b={x ≤ y}` and `c={x=y+1}`

preconditions:

b	b		b’	c’
x≤y	x=y+1		x≤y+1	x=y+2
T	T			
T	F			
F	T			
F	F			

Multiple Predicates

Two predicates: `b={x ≤ y}` and `c={x=y+1}`

preconditions:

```
{x ≤ y + 1}
y := y + 1
{x ≤ y}

{x = y + 2}
y := y + 1
{x = y+1}
```

`y:=y+1` is abstracted to

```
simultaneous
  b := b&&!c || !b&&c ? T : F
  c := b&&!c || !b&&c ? F : *
end
```

b	b		b’	c’
x≤y	x=y+1		x≤y+1	x=y+2
T	T	X		
T	F	a ≤ b	T	F
F	T	a=b+1	T	F
F	F	a>b+1	F	*

In general, simultaneous assignments are needed for abstract statements

Abstraction of Conditional

We use * to denote a nondeterministic value

b	
{x odd}	{x = 5}
T	
F	

Original Program

```
if(x == 5) then
  S1
else
  S2
fi
```

Abstract Program `b = {x odd}`

Abstraction of Conditional

We use `*` to denote a nondeterministic value

`b`	
`{x odd}`	`{x = 5}`
T	*
F	F

Original Program

```
if(x == 5) then
  S1
else
  S2
fi
```

Abstract Program (`b = {x odd}`)

```
if(b?*:F) then
  S1
else
  S2
fi
```

Note:

- b=false is the same as x even, which implies x!=5.
- b=true means that x is odd, which means x may or may not be 5

Another Example

```
done = 0;
while(done == 0){
  if(x != 0)
    x--;
  else
    done++;
}
assert(x == 0);
```

How do you argue that the program is correct?

Which predicates do you need to prove that?

Another Example

```
done = 0;
while(done == 0){
  if(x != 0)
    x--;
  else
    done++;
}
assert(x == 0);
```

Function Calls

```
int y;

f(){
   h(y);
}

void h(int z){
   y = z;
}
```

```
bool b; // y>0

f(){

}

void h(bool c){ // c: z =0

}
```

Function Calls

```
int y;

f(){
   h(y);
}

void h(int z){
   y = z;
}
```

```
bool b; // y>0

f(){
   h(b ? F : *)
}

void h(bool c){ // c: z =0
   b = c ? F : *;
}
```

Abstraction

- Tricky: find the proper abstraction!
  - You use the counterexamples, but how?
  - You can do it by hand
  - You can try to do it automatically
- Automatically finding the proper abstraction cannot always work. Why not?

Precisely: assignment

Original: `x:= e`
Predicates p1,…,pn.

Suppose we have

```
{qi}
x := e;
{pi}
```

Let ai be the disjunction of assignments to p1…pn that imply qi.

let bi be the disjunction of assignments to p1…pn that imply ¬qi.

`x := e` is replaced by

```
simultaneous
  p1 = a1 ? T : b1 ? F : *
  …
  pn = an ? T : bn ? F : *
end simultaneous
```

example

Assignment: b := b+1
Predicates: `p1 = {a ≤ b}` and `p2 = {a=b+1}`

```
{a ≤ b + 1}     {a = b + 2}
b := b + 1      b := b + 1
{a ≤ b}         {a = b + 1}
```

```
Look at the table: row TT, TF, and FT have a T in
column a≤b and TT and FF have an F in that
column.  Therefore:
p1 ∨ p2              implies a ≤ b + 1
(p1∧p2)∨(¬p1∧¬p2) implies a > b + 1
(note: false implies anything)
```

For the 2^{nd} predicate:

```
p1 ∧ p2 implies a = b+2
p1 ∨ ¬p2 implies a ≠ b+2
```

`b:=b+1` is abstracted to

```
simultaneous
 {a≤b}   := p1||p2 ? T : p1==p2 ? F : *
 {a=b+1} := p1&&p2 ? T : p1!=p2 ? F : *
end
```

p1	p2			
a≤b	a=b+1		a≤b+1	a=b+2
T	T	×	T/F	T/F
T	F	a≤b	T	F
F	T	a=b+1	T	F
F	F	a>b+1	F	*

(Cf. same example on an earlier slide)