Combining Testing and Verification for better Assertion Checking in Sequential Programs

C. Prasanth

Advisers: Dr. Aditya Nori
Dr. M. V. Panduranga Rao

Outline

- Overview of Symbolic execution
- Overview of Dynamic Automated Random Testing (DART)
- Synergy and its working on examples
- DASH and its working on examples
- SMASH and its working on examples
- Problem statement and Proposed solution
- Conclusion

Assertion Checking

Given

a sequential program P with inputs I and
an assertion "assert(e)"

Questions:

Bug finding: Does there exist an execution of the program P for some input I such that the assertion is violated?

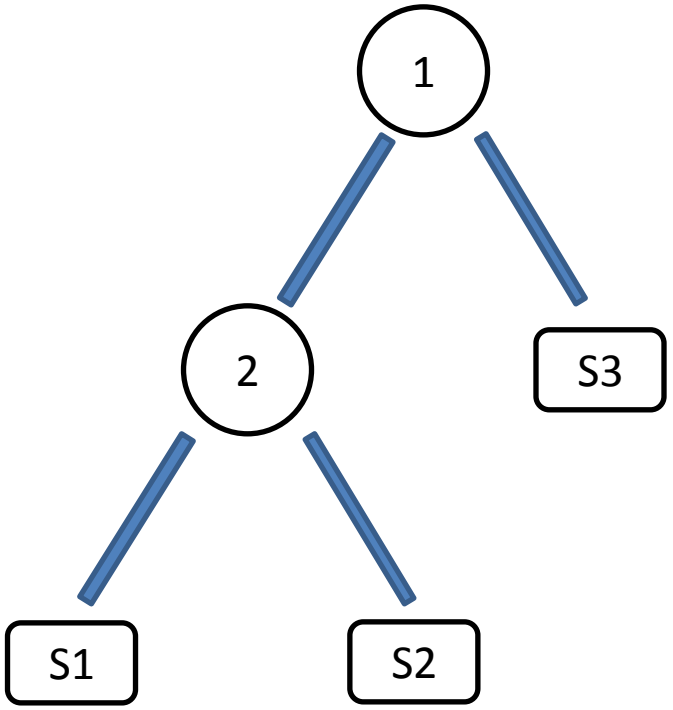
Verification: Does the assertion hold for all possible inputs?

Overview of Symbolic execution

- Notion of concrete inputs
- Notion of symbolic inputs
- Notion of state values in symbolic form
- Symbolically executing each program statement
    - Using symbolic inputs
- Notion of path condition
    - Characterization of a path
    - Notion of deterministic and non deterministic if's

J. C. King. Symbolic Execution and Program Testing. Communications. ACM 19, (1976) 385–394.

```
int foo (int x, int y ) {

    if ( x > 10 ) {                1

        if ( y < 10 )              2

            print “path 1”;  x =1; y = 1;   S1

        else

            print “path2”; x = 1; y = 2;  S2

    else

        print “path3”;   x = 2 ; y = 0;    S3
}
```



```
int foo (int x, int y ) {
    if ( x > 10 ) {                1
        if ( y < 10 )              2
            print “path 1”;  x =1; y = 1;   S1
        else
            print “path2”; x = 1; y = 2;  S2
    else
        print “path3”;   x = 2 ; y = 0;    S3
}
```

Symbolic state (Sx, Sy)

Path condition

```
int foo (int x, int y ) {

    if ( x > 10 ) {                1

        if ( y < 10 )              2

            print “path 1”;  x =1; y = 1;   S1

        else

            print “path2”; x = 1; y = 2;  S2

    else

        print “path3”;   x = 2 ; y = 0;    S3
}
```

Symbolic state (Sx, Sy)

Path condition

Sx > 10

```
int foo (int x, int y ) {

    if ( x > 10 ) {              1

        if ( y < 10 )            2

            print “path 1”;  x =1; y = 1;   S1

        else

            print “path2”; x = 1; y = 2;  S2

    else

        print “path3”;   x = 2 ; y = 0;    S3
}
```

Symbolic state (Sx, Sy)

Path condition

Sx > 10

Sy < 10

```
int foo (int x, int y ) {

    if ( x > 10 ) {              1

        if ( y < 10 )            2

            print “path 1”;  x =1; y = 1;   S1

        else

            print “path2”; x = 1; y = 2;  S2

    else

        print “path3”;   x = 2 ; y = 0;    S3
}
```

Symbolic state (Sx, Sy)

Sx = 1, Sy = 1

Path condition

Sx > 10

Sy < 10

```
int foo (int x, int y ) {

    if ( x > 10 ) {                1

        if ( y < 10 )              2

            print “path 1”;  x =1; y = 1;   S1

        else

            print “path2”; x = 1; y = 2;  S2

    else

        print “path3”;   x = 2 ; y = 0;    S3
}
```

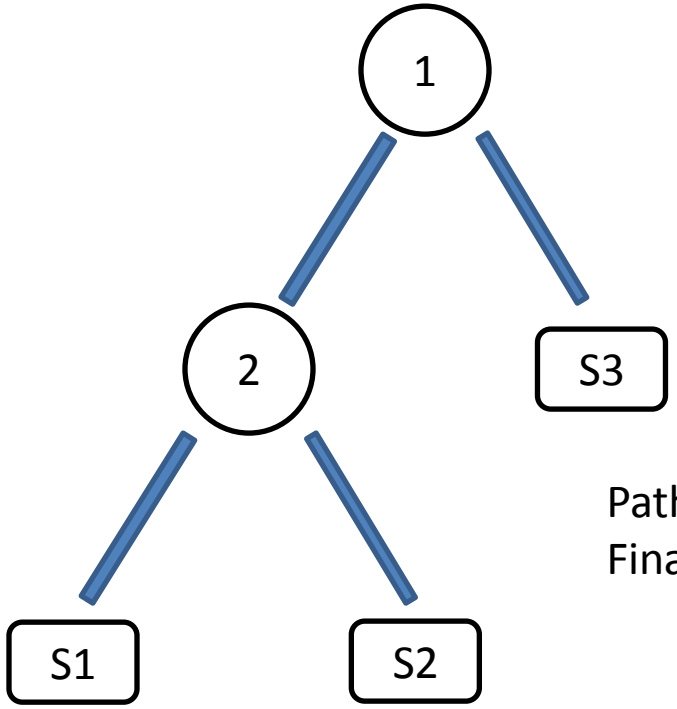
Symbolic state (Sx, Sy)

Sx = 1, Sy = 1

Path condition

Sx > 10

Sy < 10



Path condition: ~ (Sx > 10)
Final state: Sx = 2, Sy = 0

PC: Sx > 10 and Sy < 10
Final state: Sx = 1, Sy = 1

PC: Sx > 10 and ~(Sy < 10)
Final state: Sx = 1, Sy = 2

Idea !!!

- One immediate idea for assertion checking would be "Execute all program paths symbolically and compare the assertion with results of symbolic execution of all program paths. If any one matches, give path condition of matched path to constraint solver to get feasible inputs".

- Drawback: Limitations of symbolic execution
  - Solves only linear constraints
  - Limitations of inter functional analysis
  - If the called function code is not available to analyze

Outline

- Overview of Symbolic execution [DONE]
- Overview of Dynamic Automated Random Testing (DART)
- Synergy and its working on examples
- DASH and its working on examples
- SMASH and its working on examples
- Problem statement and Proposed solution
- Conclusion

Overview of DART

Dynamic test generation consists of

1. Executing the program P, starting with some given or random inputs

2. Gathering symbolic constraints on inputs at conditional statements along the execution and

3. Using a constraint solver to infer variants of the previous inputs to steer the programs next execution toward an alternative program branch.

This process is repeated until an error statement is reached.

Whenever symbolic execution does not know how to generate a constraint for a program statement depending on some inputs, that constraint will be simplified using those inputs concrete values at that point of execution.

Example

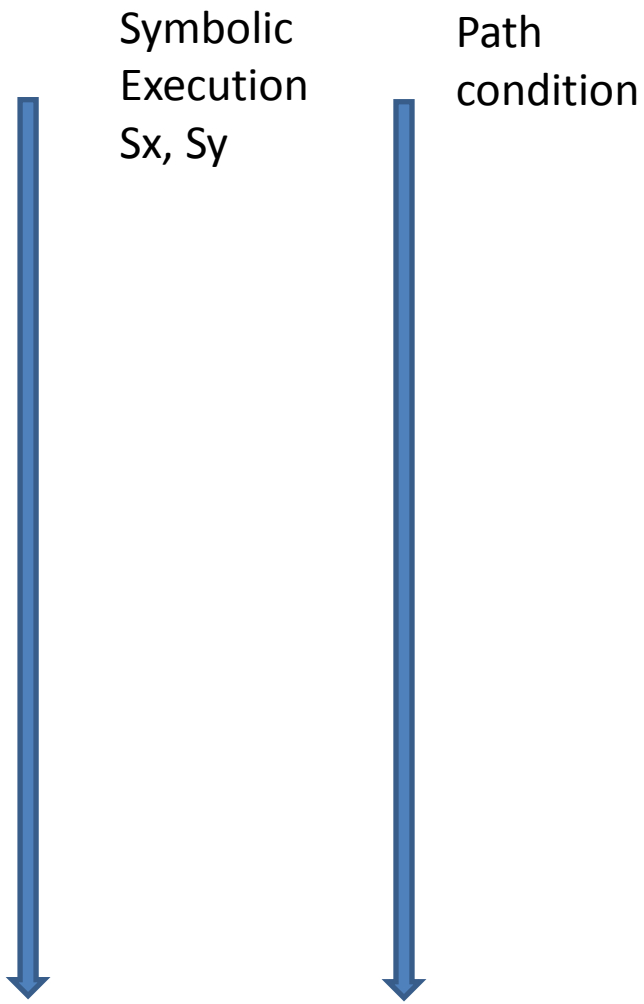
Consider following program.

```
int foo (int x, int y) {
    if ( y == hash(x) )
        error ;
    else
        print “No bugs”;
}
```

Example

Consider following program.

```
int foo (int x, int y) {

    if ( y == hash(x) )

        error ;

    else

        print “No bugs”;

}
```



Run 1: Random Inputs (10, 42)

Consider following program.

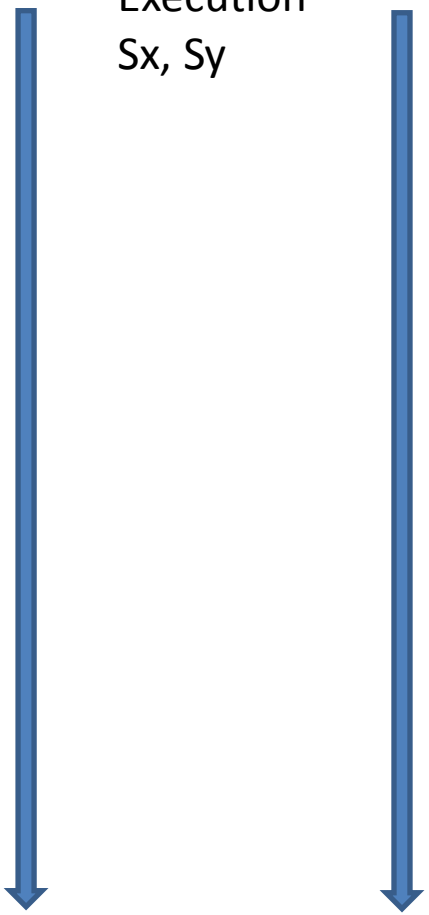
```
int foo (int x, int y) {

    if ( y == hash(x) )

        error ;

    else

        print “No bugs”;

}
```

Let hash (10) = 23

Concrete Execution

Symbolic Execution
Sx, Sy

Path condition



Run 1: Random Inputs (10, 42)

Consider following program.

```
int foo (int x, int y) {

    if ( y == hash(x) )

        error ;

    else

        print “No bugs”;

}
```

Let hash (10) = 23

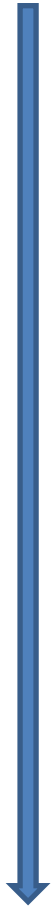
Concrete Execution

42 != 23

Symbolic Execution
Sx, Sy

Path condition

Negation
Sy= hash(Sx)



Run 1: Random Inputs (10, 42)

Consider following program.

```
int foo (int x, int y) {

    if ( y == hash(x) )

        error ;

    else

        print “No bugs”;

}
```

Concrete Execution

42 != 23

Symbolic Execution Sx, Sy

Path condition

~(Sy = 23)

Since hash(Sx) cannot be reasoned symbolically, it will be replaced by concrete value of hash (Sx)

Run 1: Random Inputs (10, 42)

Consider following program.

```
int foo (int x, int y) {

    if ( y == hash(x) )
        error ;
    else
        print “No bugs”;
}
```

Concrete Execution

42 != 23

x = 10, y = 42

Symbolic Execution Sx, Sy

Path condition

~(Sy = 23)

After solving constraints, input values are 10, 23

Run 2: Directed Inputs (10, 23)

Consider following program.

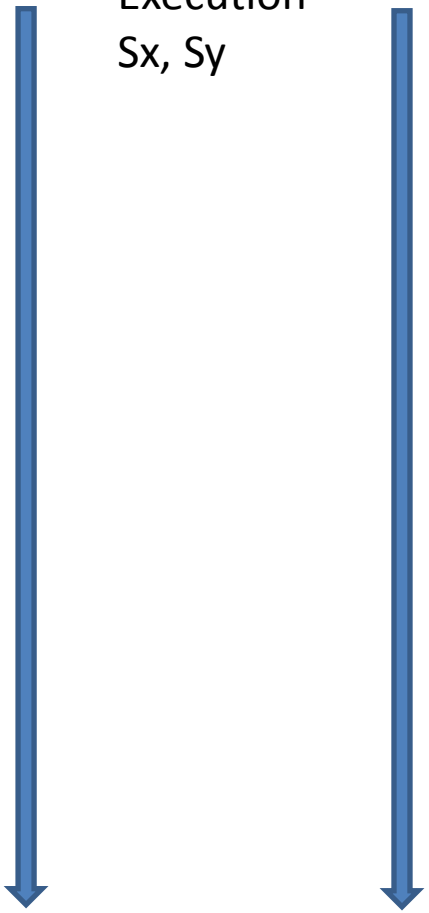
```
int foo (int x, int y) {

    if ( y == hash(x) )

        error ;

    else

        print “No bugs”;

}
```

Let hash (10) = 23

Concrete Execution

Symbolic Execution Sx, Sy

Path condition



Run 2: Directed Inputs (10, 23)

Consider following program.

```
int foo (int x, int y) {

    if ( y == hash(x) )
        error ;

    else
        print “No bugs”;

}
```

Let hash (10) = 23

Concrete Execution

23 == 23

Symbolic Execution
Sx, Sy

Path condition

Sy= hash(Sx)
--
Sy = 23

Run 2: Directed Inputs (10, 23)

Consider following program.

```
int foo (int x, int y) {

    if ( y == hash(x) )

        error ;

    else

        print “No bugs”;

}
```

After solving constraints, input values are 10, 23

Concrete Execution

42 != 23

x = 10, y = 23

Symbolic Execution

Sx, Sy

Path condition

Sy= hash(Sx)

--

Sy = 23

Idea !!

- Why are we starting with random inputs and moving towards a path that leads to assertion ??
  - We don't know where assertion is put !!

- What if we know path that leads to assertion initially ??
  - Do symbolic execution on that path and get feasible inputs through constraint solver

- How do we know that path ??
  - Using control flow graph / Abstraction of a program

Idea !!

- How far should we believe the path from abstraction ??
  - while loops
  - Infeasible if conditions

- Remove the spurious paths from abstraction to get correct path
  - Refine the abstraction

- How far should we go to remove spurious paths from abstraction??
  - Either have a concrete input that leads to assertion [or]
  - There is no path given by abstraction

Idea !!

- Is there any chance where abstraction doesn't give a path and there exists a concrete inputs that leads to assertion ??
  - No
- Abstraction is always the over approximation of given control flow of a program.
- If Abstraction doesn't satisfies some property, then concrete program also doesn't satisfy that property.
- If Abstraction satisfy some property, then concrete may/ may not satisfy that property.

Synergy Idea !!

- How do we refine Abstraction ??
  - Spurious counter example guided refinement
  - Using test cases to refine abstract states

- Refining Abstraction as directed by test cases.

- Finding path to error as directed by abstraction.

- Synergizing both abstraction and test cases for better assertion checking -- SYNERGY Algorithm

Outline

- Overview of Symbolic execution [DONE]
- Overview of (DART) [DONE]
- Synergy and its working on examples
- DASH and its working on examples
- SMASH and its working on examples
- Problem statement and Proposed solution
- Conclusion

Overview of SYNERGY Algorithm

Combining Testing and Verification by above ideas

Given

a sequential program P with inputs I (say, written in C)

an assertion "assert(e)"

Answer from SYNERGY:

Pass Result: (Bug) : Witness is error trace

Fail Result: (Proof) : Finite Indexed partition of abstraction

Non terminating: Since verification question is undecidable.

- Maintains two data structures
  - Abstraction (Proof) – Finite indexed partition
  - Forest (Tests) - Set of execution paths from test cases

- Notion of Abstract error trace
  - Path from initial node to error node in Abstraction

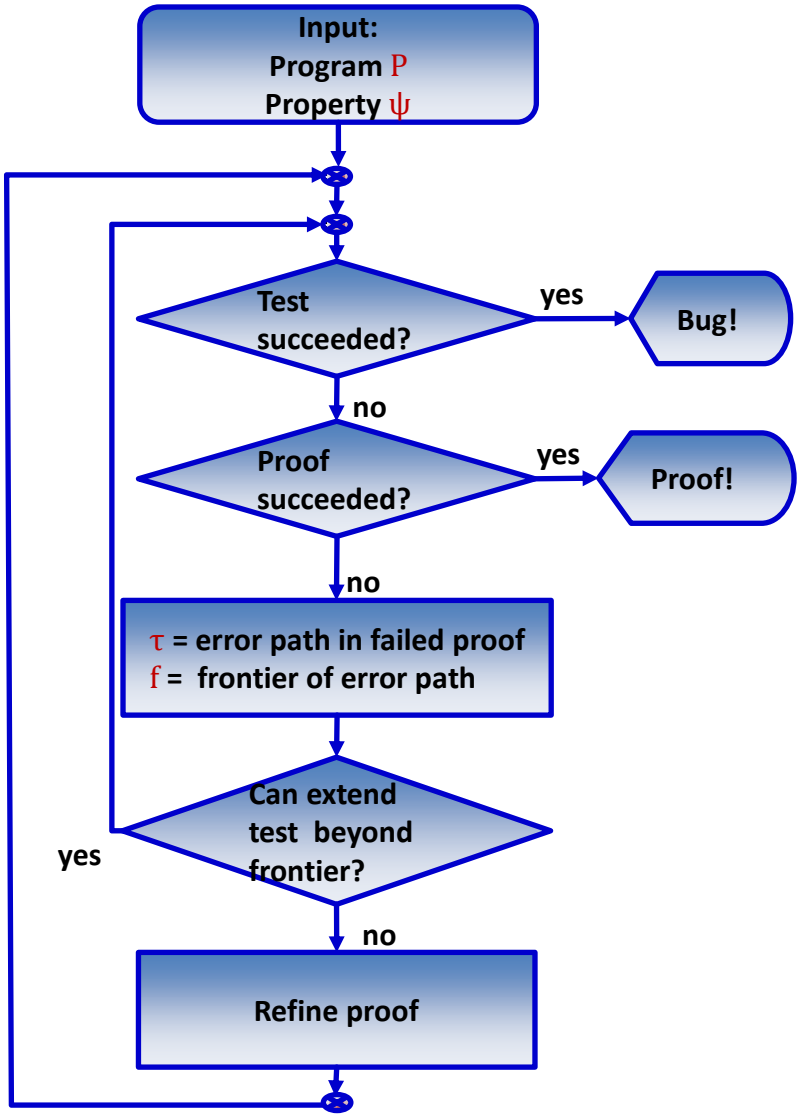
- Notion of Concrete error trace
  - Path from initial node to error node in Forest

- Notion of Frontier
  - Unvisited node by concrete execution in Abstract error trace

- Notion of Refinement

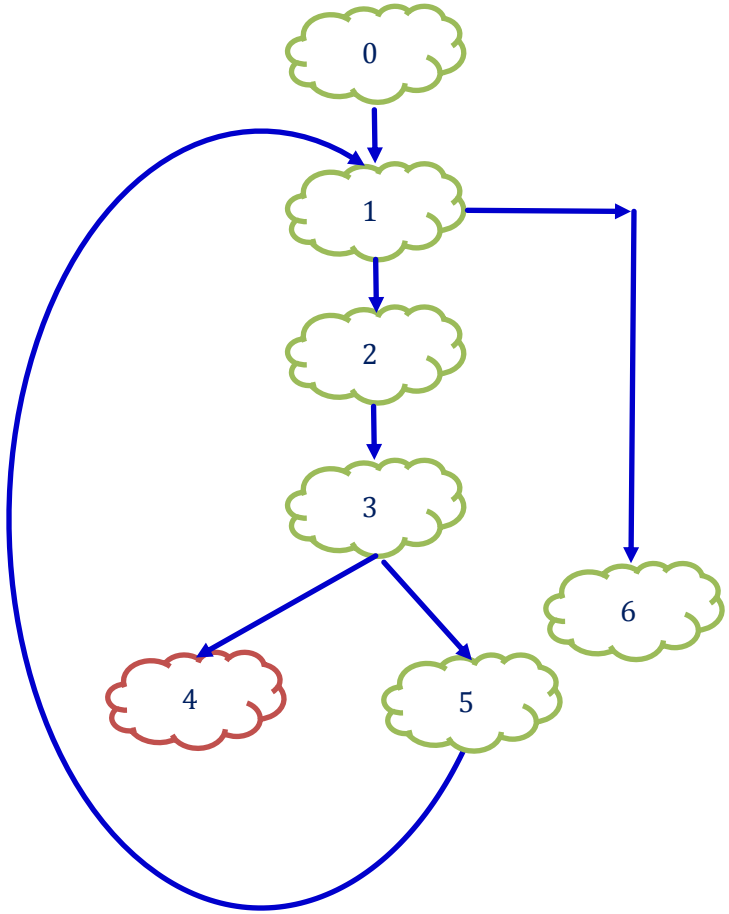
- Using preImage operator
  - Getting pre condition from statement and post condition
  - Ex: {pre} {x = y} { x > 10}
  - Precondition would be pre = { y > 10}

Synergy sketch



Example -1 (Proof case)

```
   void  foo (int n) {
0:  int i = 0;
1:  while ( i < n ) {
2:       int a = i;
3:       if ( a >= n )
4:            error ();
5:       i++; }
6: return ; }
```

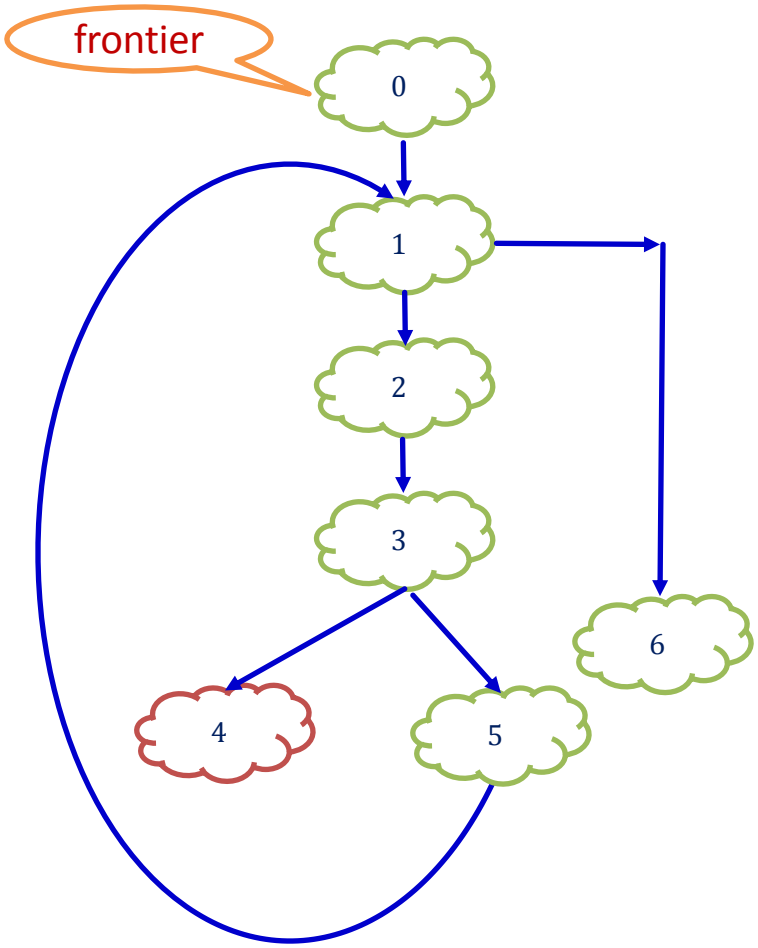


Iteration - 1

Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 4} and
Frontier is 0



Iteration - 1

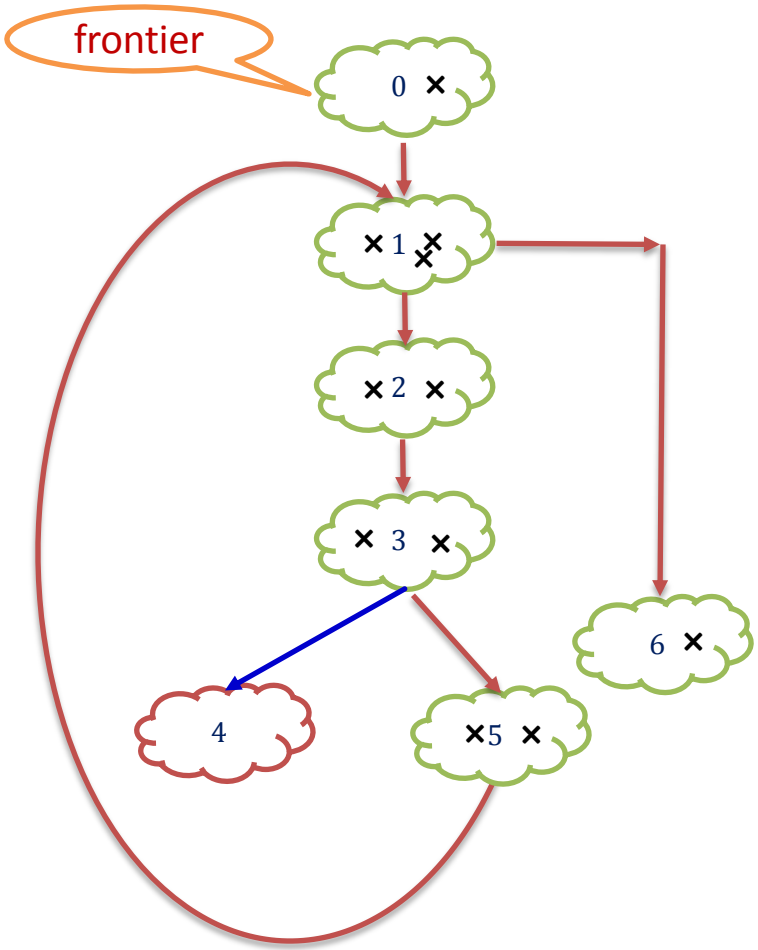
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 4} and
Frontier is 0

Step4: Generate suitable test as n = 2

Step5: Forest F gets updating by adding this current concrete trace.



Iteration - 2

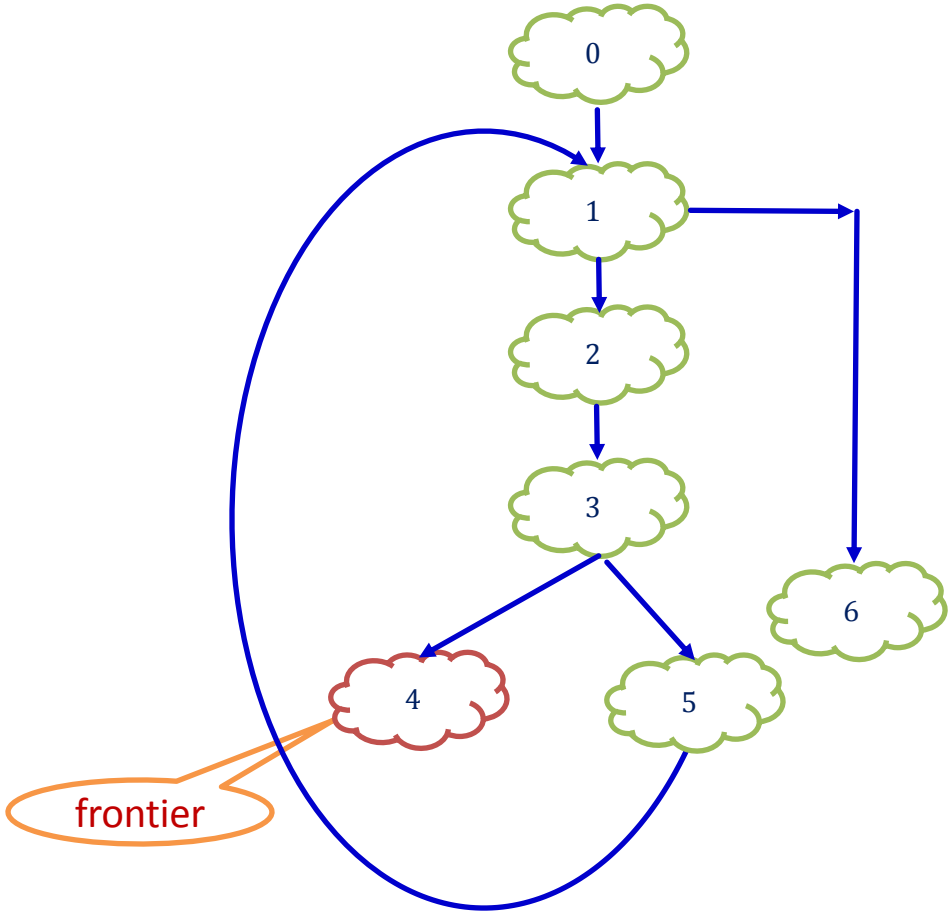
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 5, 1, 2, 3, 4}
and Frontier is 4.

Possible Abstract error traces are
{0, 1, (2, 3 , 5, 1)*, 2, 3, 4}

But It will take maximally corresponding prefix in Forest.



Iteration - 2

Step1: Couldn't get witness

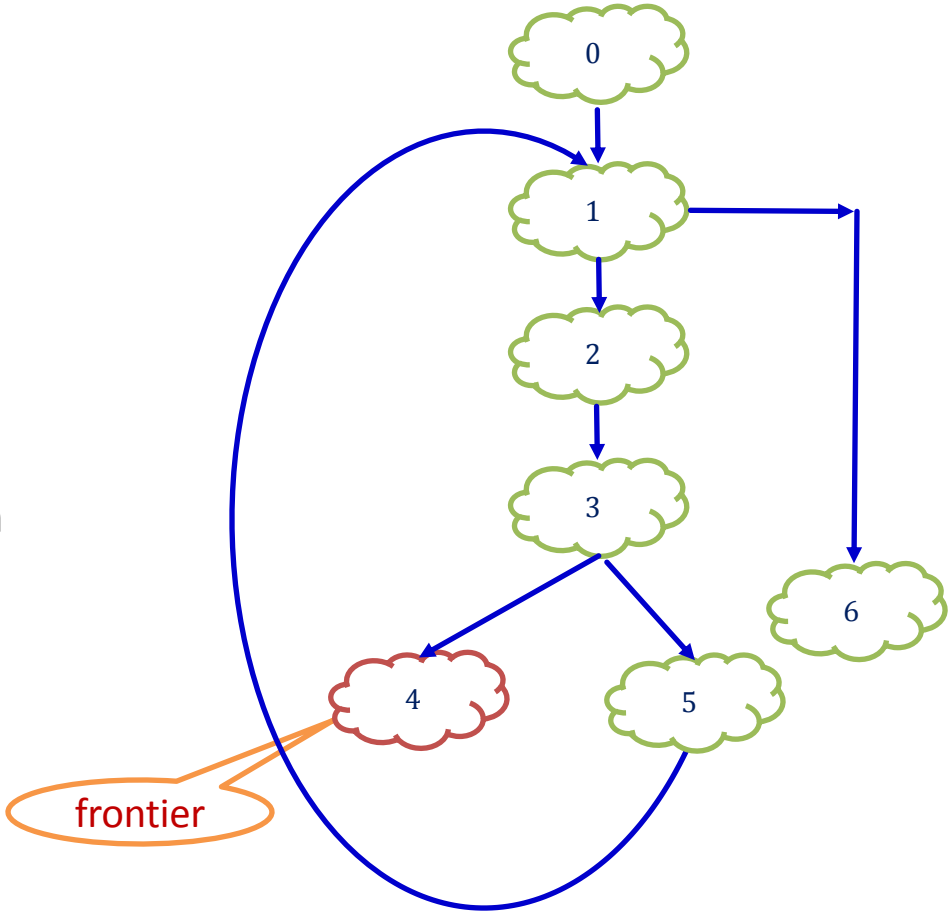
Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 5, 1, 2, 3, 5, 1, 6}
and Frontier is 4.

Step4: couldn't generate an input to reach 4 from 0.

Let N be symbolic name for variable n.

Constraints for reaching 3 from 0 through earlier path are: (i < N). But value of i is 0 at that instant in earlier path.
(0 < N)



Iteration - 2

Step1: Couldn't get witness

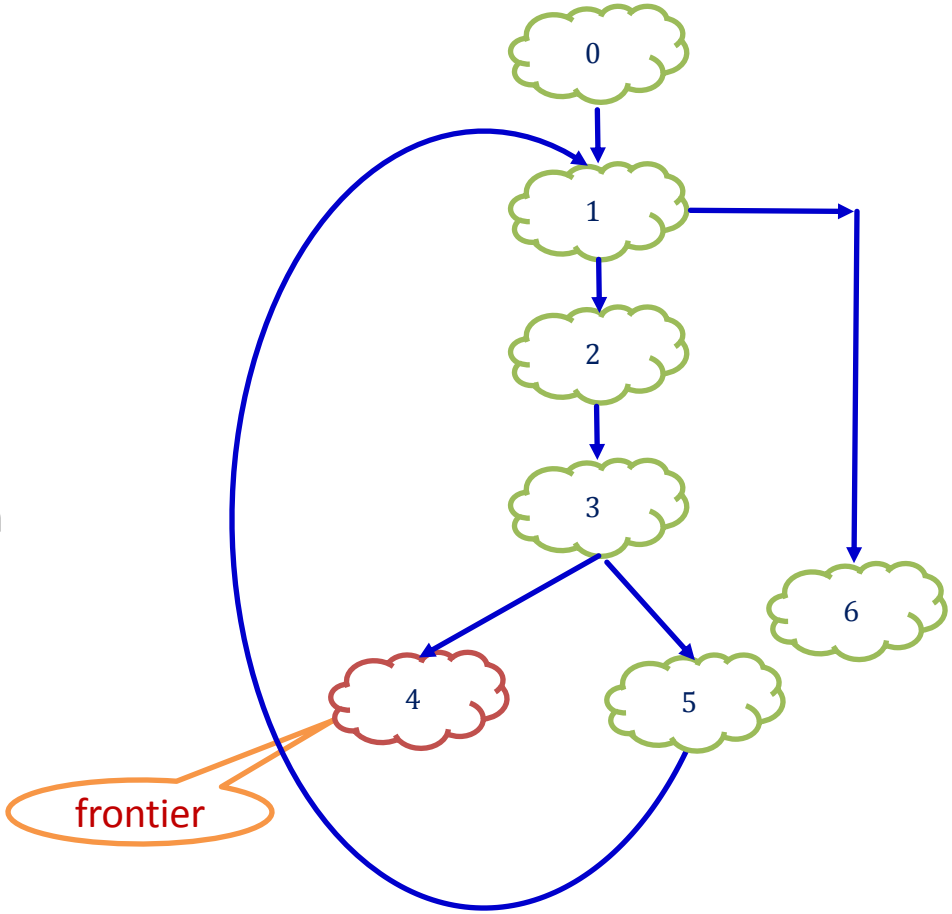
Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 5, 1, 2, 3, 5, 1, 6}
and Frontier is 4.

Step4: couldn't generate an input to reach 4 from 0.

Let N be symbolic name for variable n.

Constraints for reaching 4 from 3 is:
(a >= N). But value of a is 0 (Earlier path).
(0 >= N)



Iteration - 2

Step1: Couldn't get witness

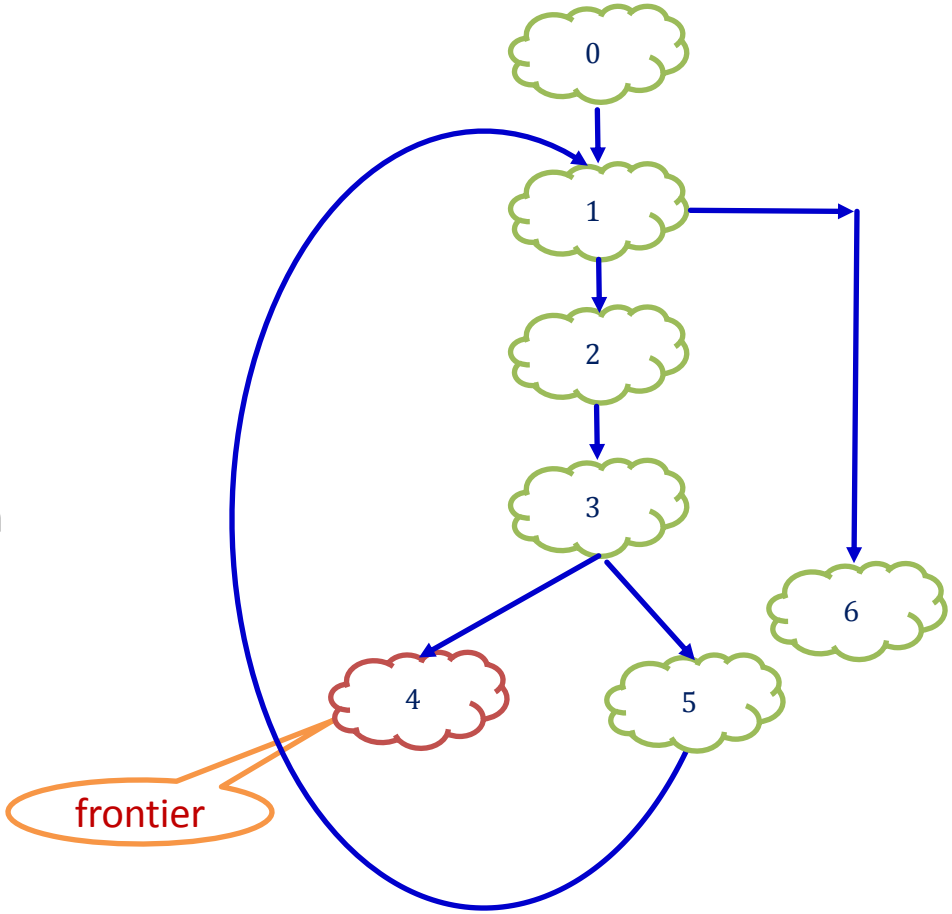
Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 5, 1, 2, 3, 5, 1, 6}
and Frontier is 4.

Step4: couldn't generate an input to reach 4 from 0.

Let N be symbolic name for variable n.
Total constraints are:

(N > 0) ^ (N <= 0) ➔ UNSAT



Iteration - 2

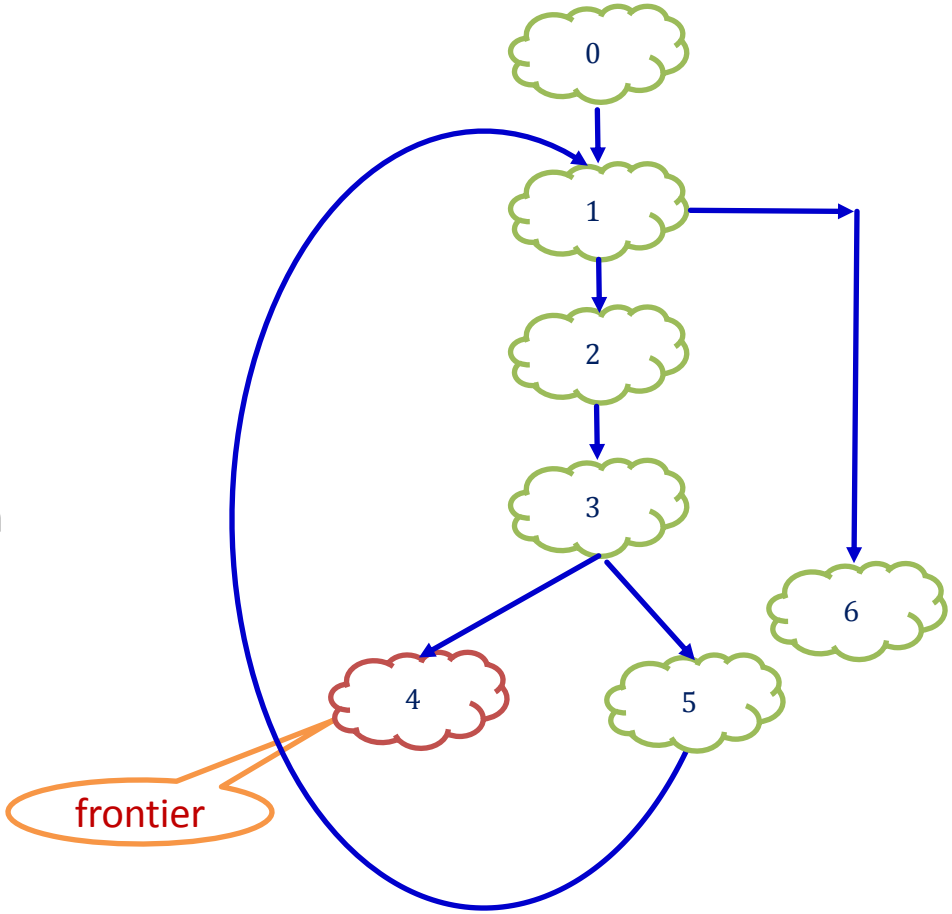
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 5, 1, 2, 3, 5, 1, 6}
and Frontier is 4.

Step4: couldn't generate an input to reach 4 from 0.

Refining state 3 into 3^P and 3^~P
Where P = (a >= N)



Iteration - 2

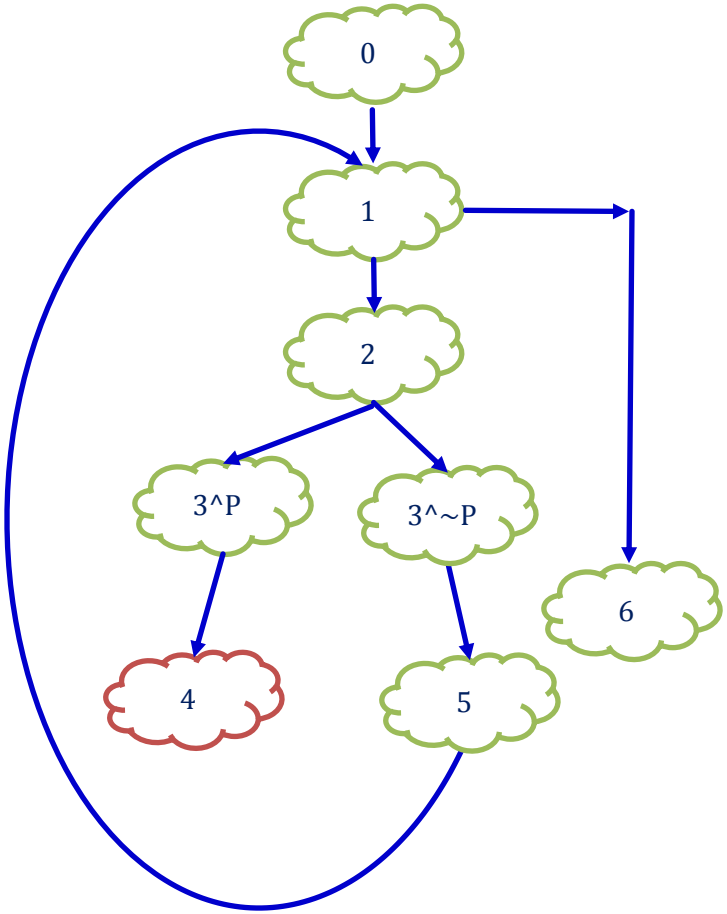
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 5, 1, 2, 3, 5, 1, 6}
and Frontier is 4.

Step4: couldn't generate an input to reach 4 from 0.

Refining state 3 into 3^P and 3^~P
Where P = (a >= N)

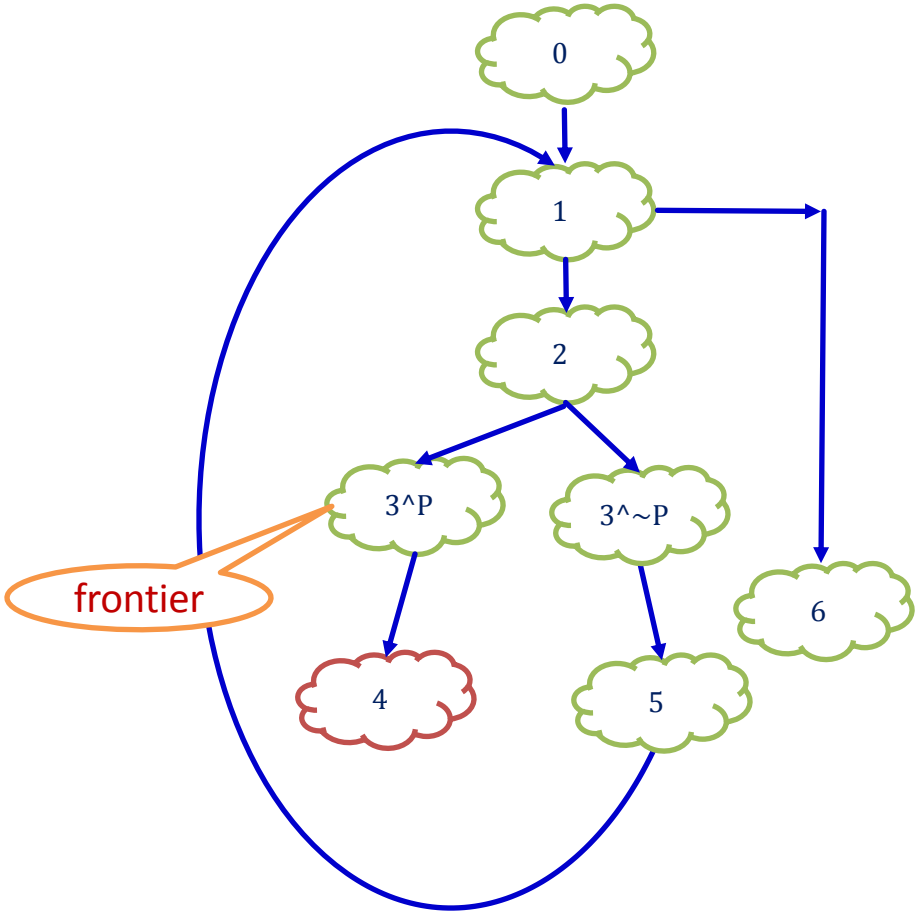


Iteration - 3

Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3^P, 4}
and Frontier is 3^P.



Iteration - 3

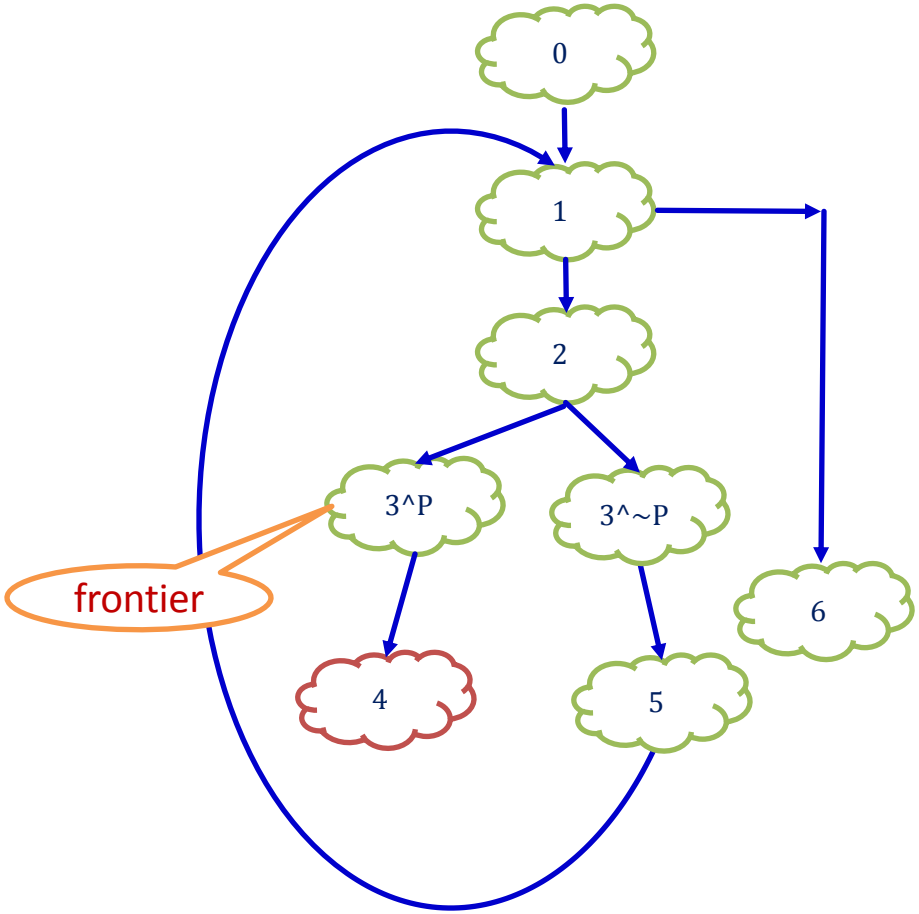
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3^P, 4}
and Frontier is 3^P.

Step4: couldn't generate an input to reach 3^P from 0.

The pre condition is Q where
Q { a = i } a >= N and it leads to Q as (i >= N)



Iteration - 3

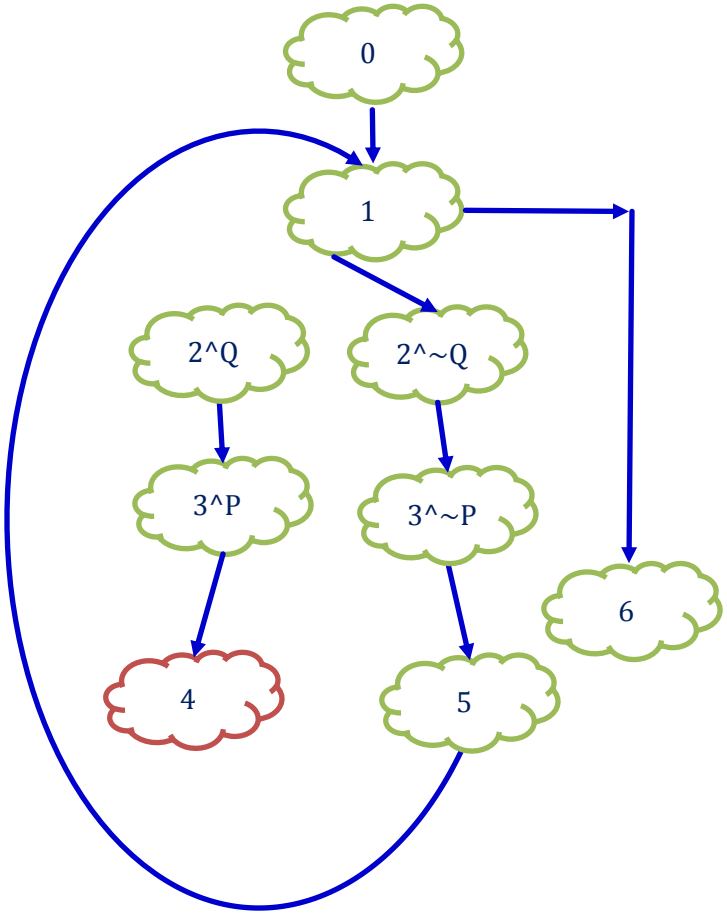
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3^P, 4}
and Frontier is 3^P.

Step4: couldn't generate an input to reach 3^P from 0.

The pre condition is Q where
Q { a = i } a >= N and it leads to Q as (i >= N)



Iteration - 3

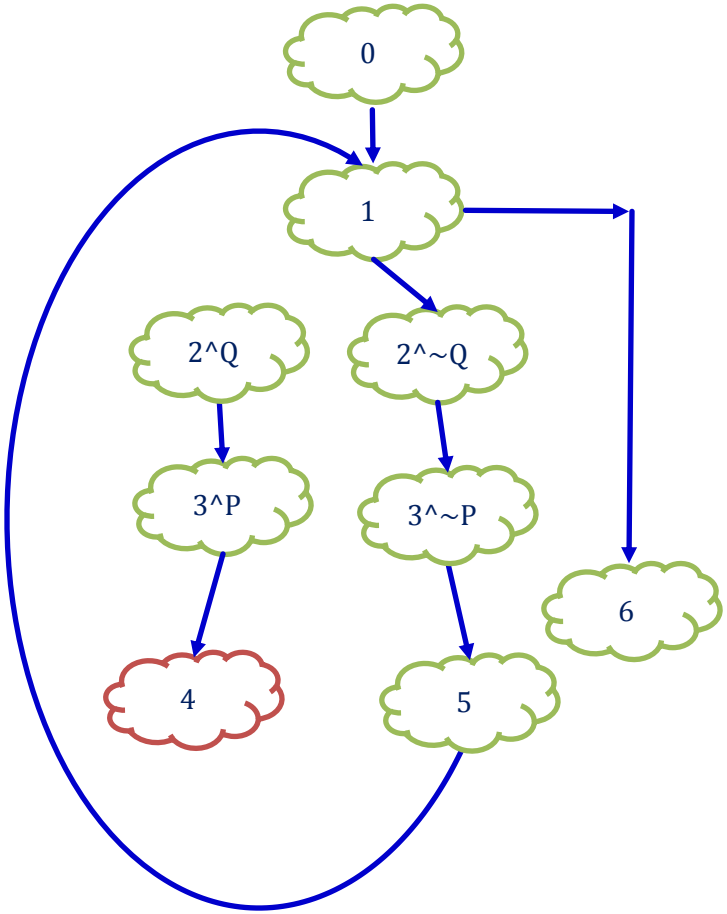
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3^P, 4}
and Frontier is 3^P.

Step4: couldn't generate an input to reach 3^P from 0.

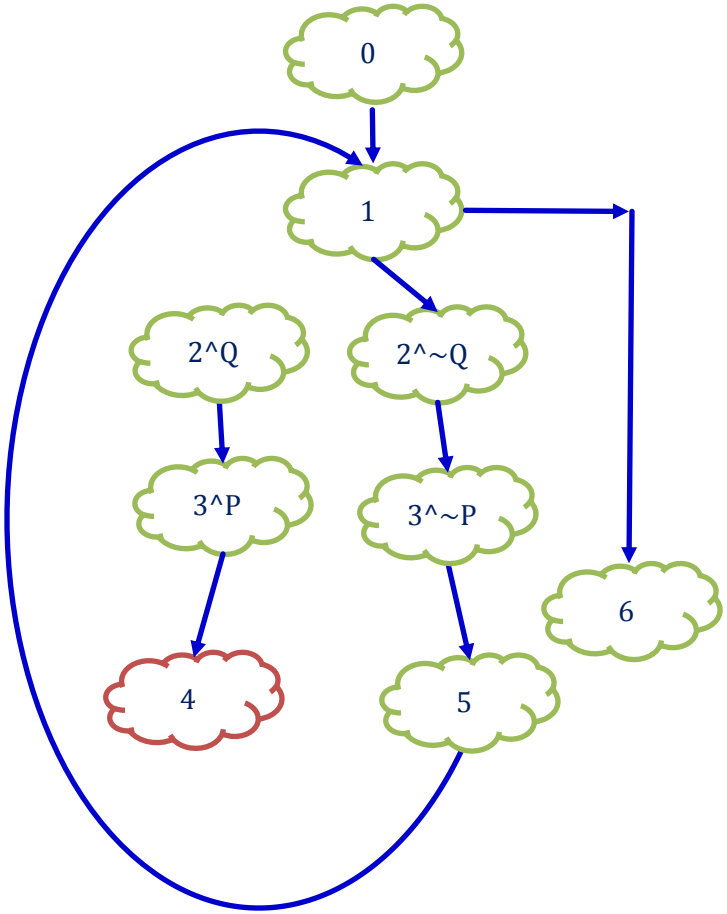
2^Q is not reachable from 1.
In order to enter loop, i < N
but Q is i >= N



Iteration - 4

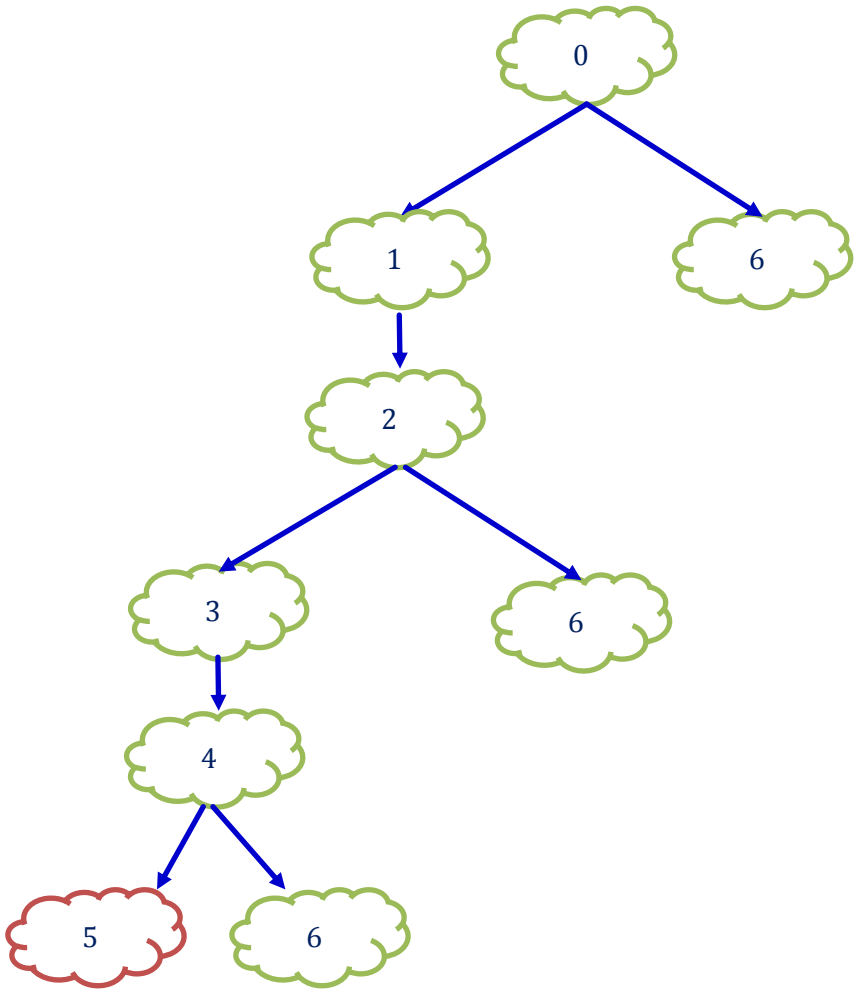
Step1: Couldn't get witness

Step2: But got the proof because synergy couldn't find an abstract error trace.



Example -2 (Test case)

```
   void foo( int a) {
0:  if ( a > 3 ) {
1:     a = a – 3;
2:     if ( a < 3 ) {
3:        a = a + 3 ;
4:         if ( a == 5 )
5:             error ();
        }
     }
   }
6:
```

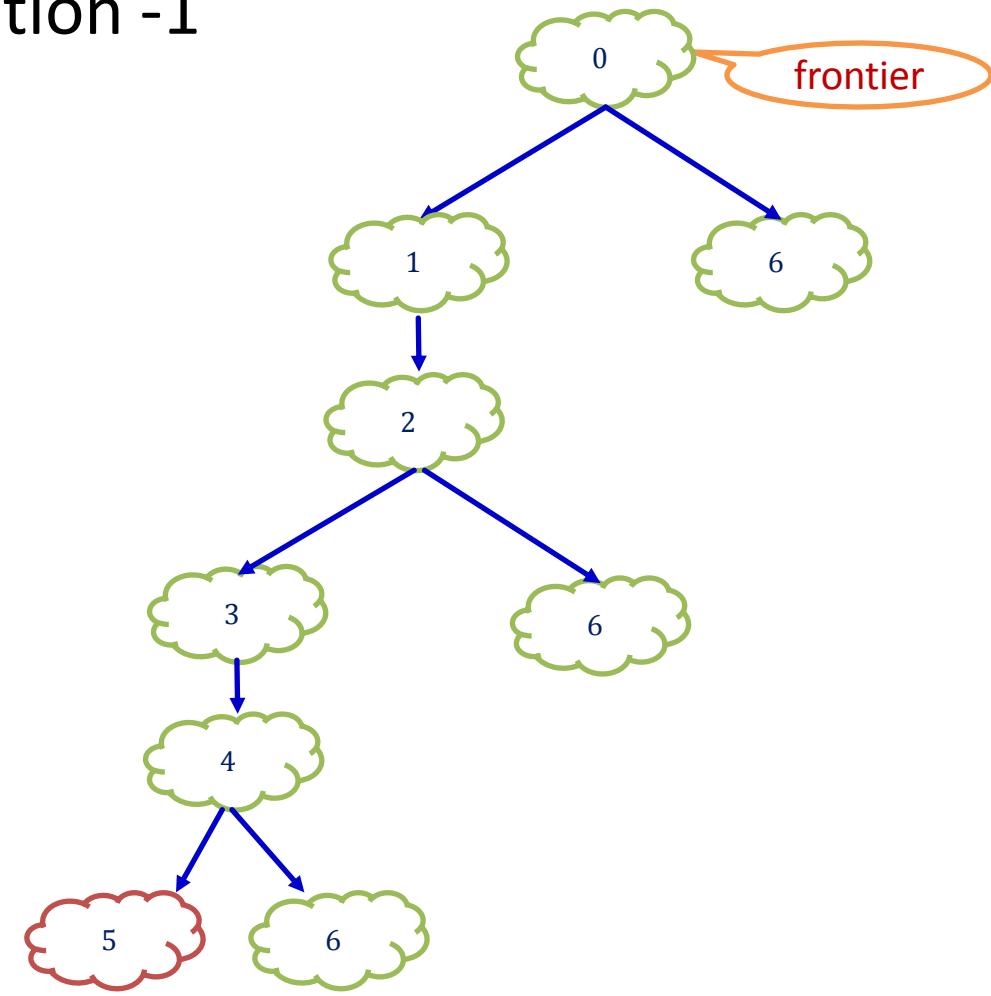


Iteration -1

Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 4, 5} and Frontier is 0



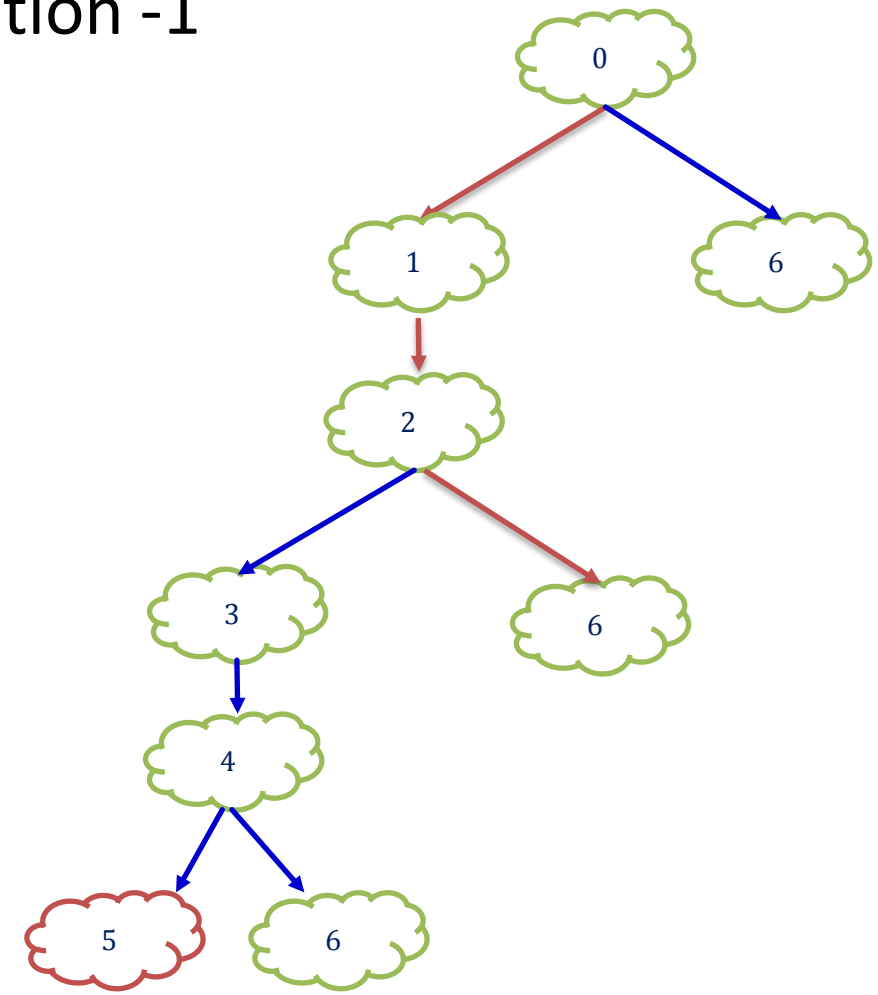
Iteration -1

Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 4, 5} and Frontier is 0

Step4: Suitable test is a = 10 and Forest F gets updated.

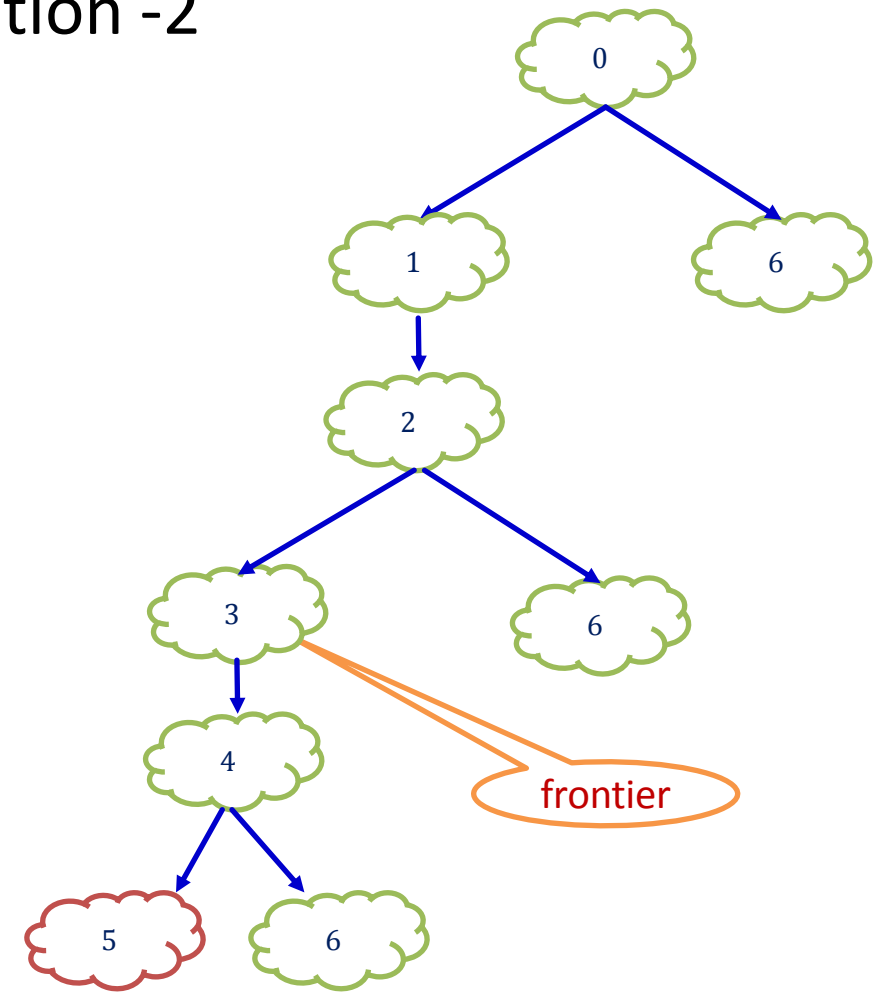


Iteration -2

Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1, 2, 3, 4, 5,} and Frontier is 3.



Iteration -2

Step1: Couldn't get witness

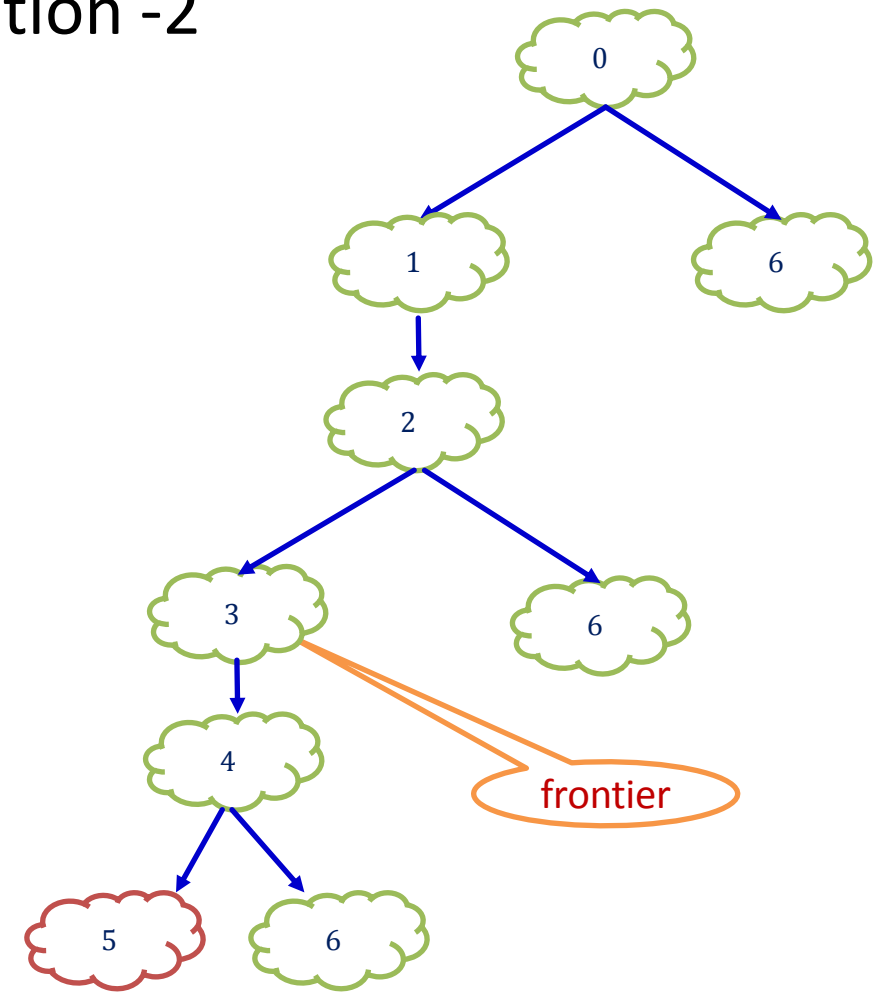
Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 4, 5,} and
Frontier is 3.

Step4:
Let A be symbolic variable for a.

Constraints from 0 to 2 are : A > 3
Constraints to reach from 2 to 3 are:
(A-3) < 3 ➔ A < 6.

Total constraints are (A > 3) ^ (A < 6)



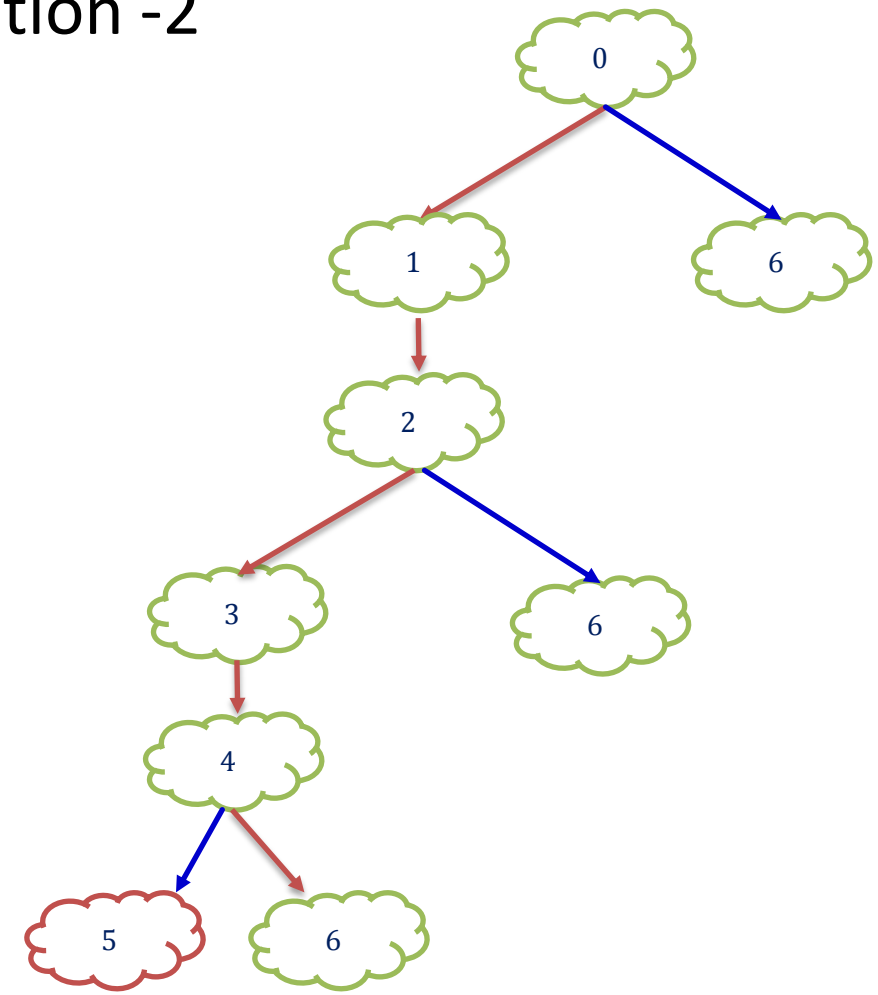
Iteration -2

Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 4, 5,} and
Frontier is 3.

Step4:
Suitable test be a = 4.

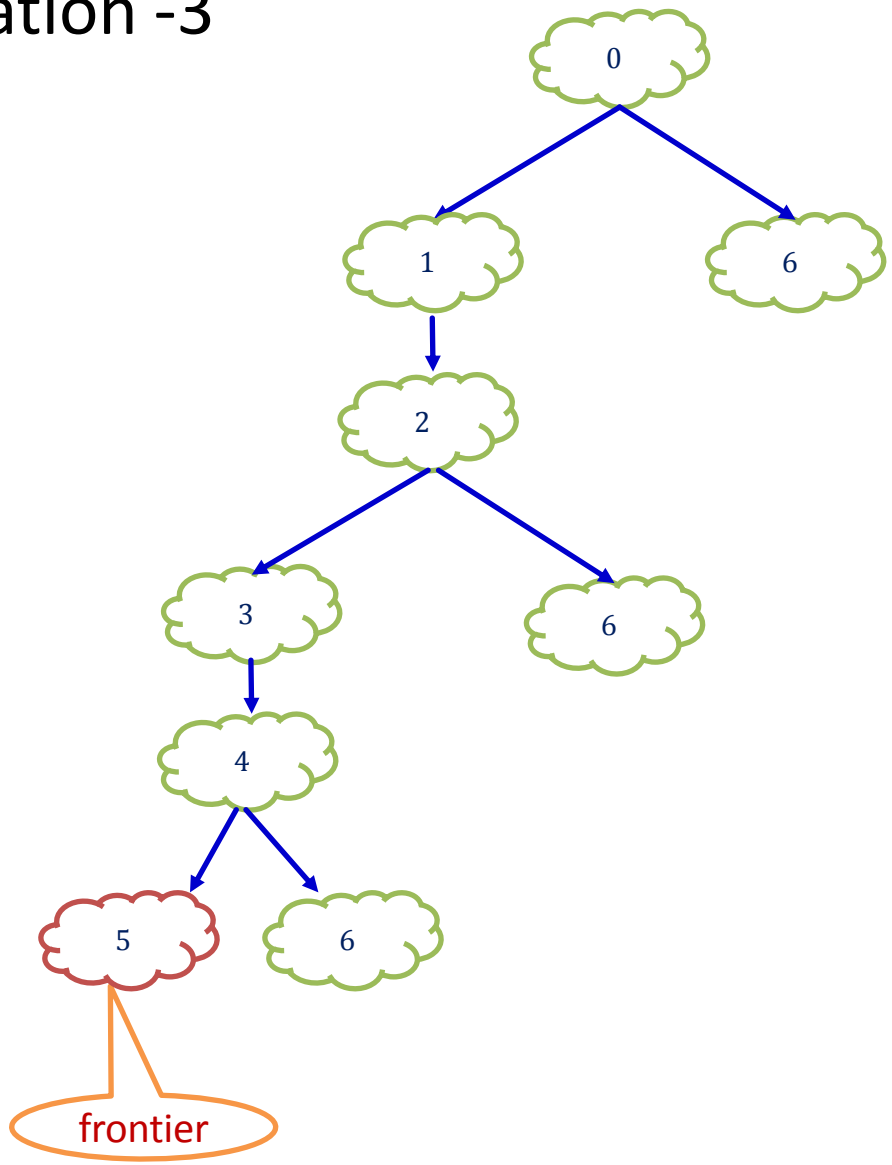


Iteration -3

Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1, 2, 3, 4, 5,} and Frontier is 5.



Iteration -3

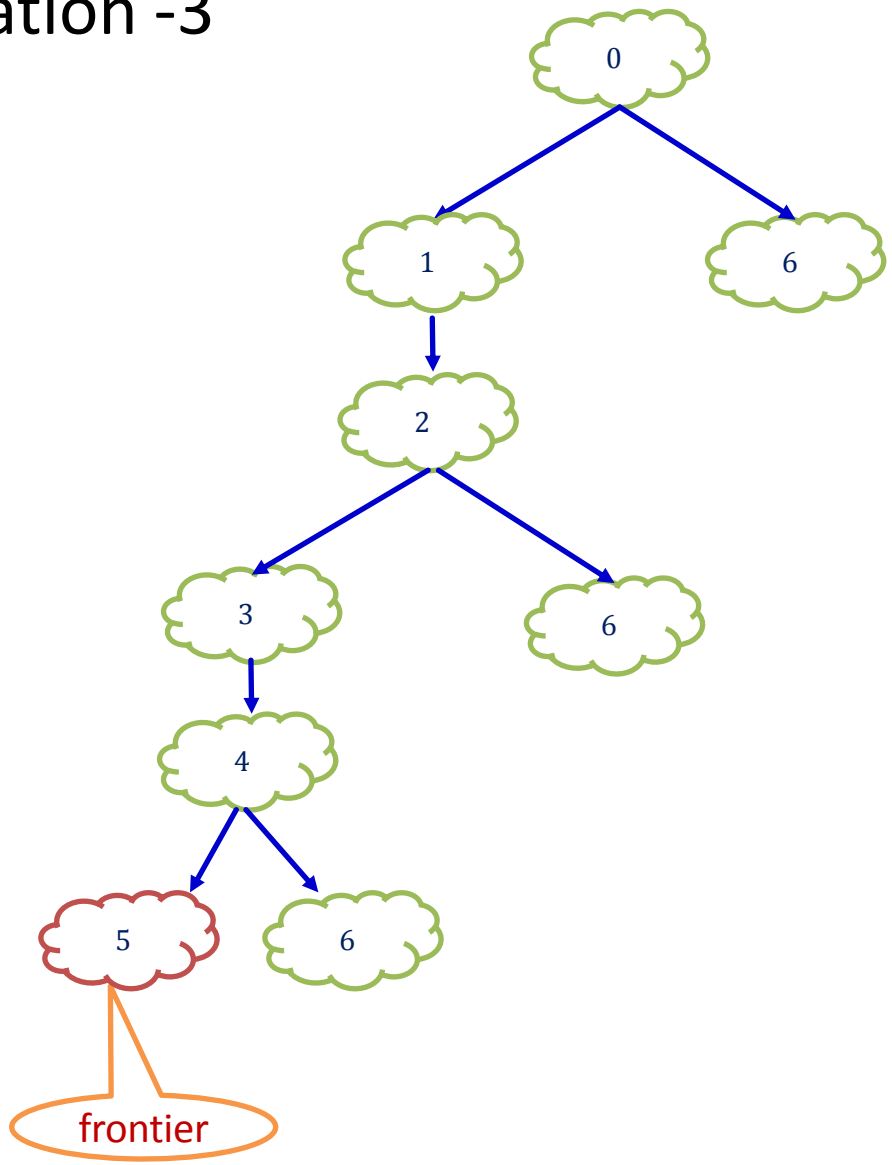
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 4, 5,} and
Frontier is 5.

Step4:
Let A be symbolic variable for a.

Constraints from 0 to 4 are :
(A > 3) ^ (A < 6)
Constraints to reach from 4 to 5 are:
(A-3+3) = 5 ➔ A = 5.
Total constraints are (A > 3) ^ (A < 6)
^ (A = 5)



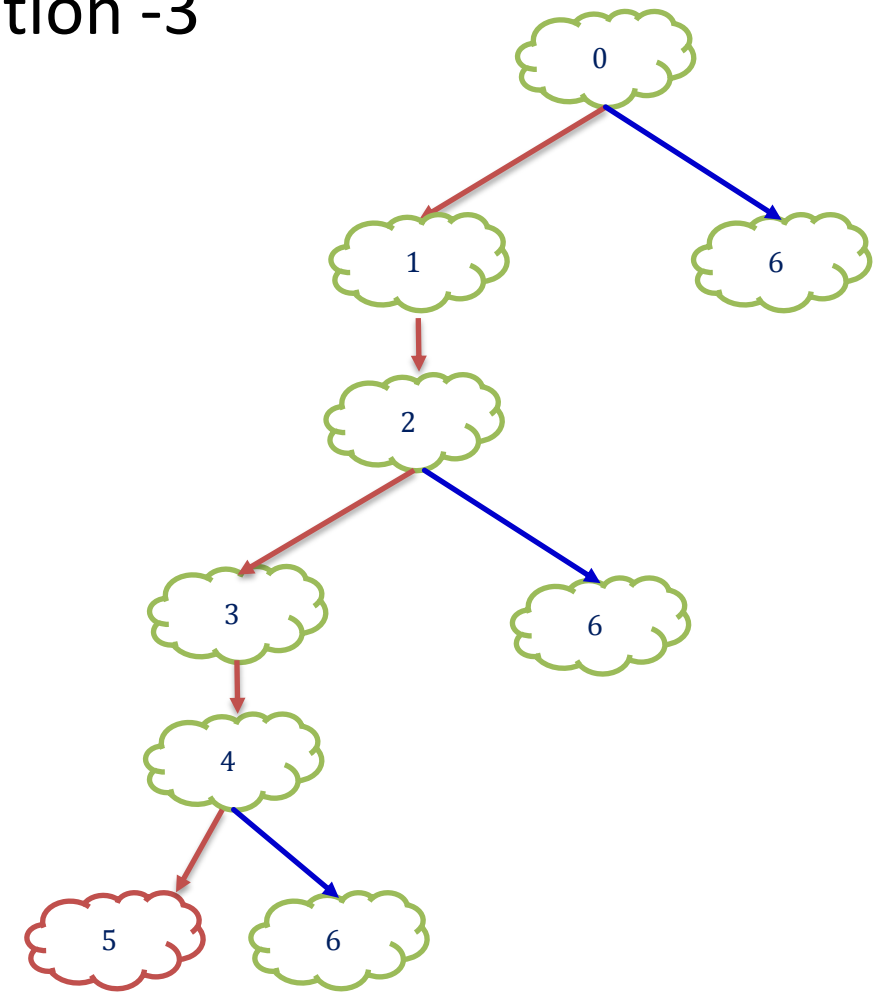
Iteration -3

Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is
{0, 1, 2, 3, 4, 5,} and
Frontier is 5.

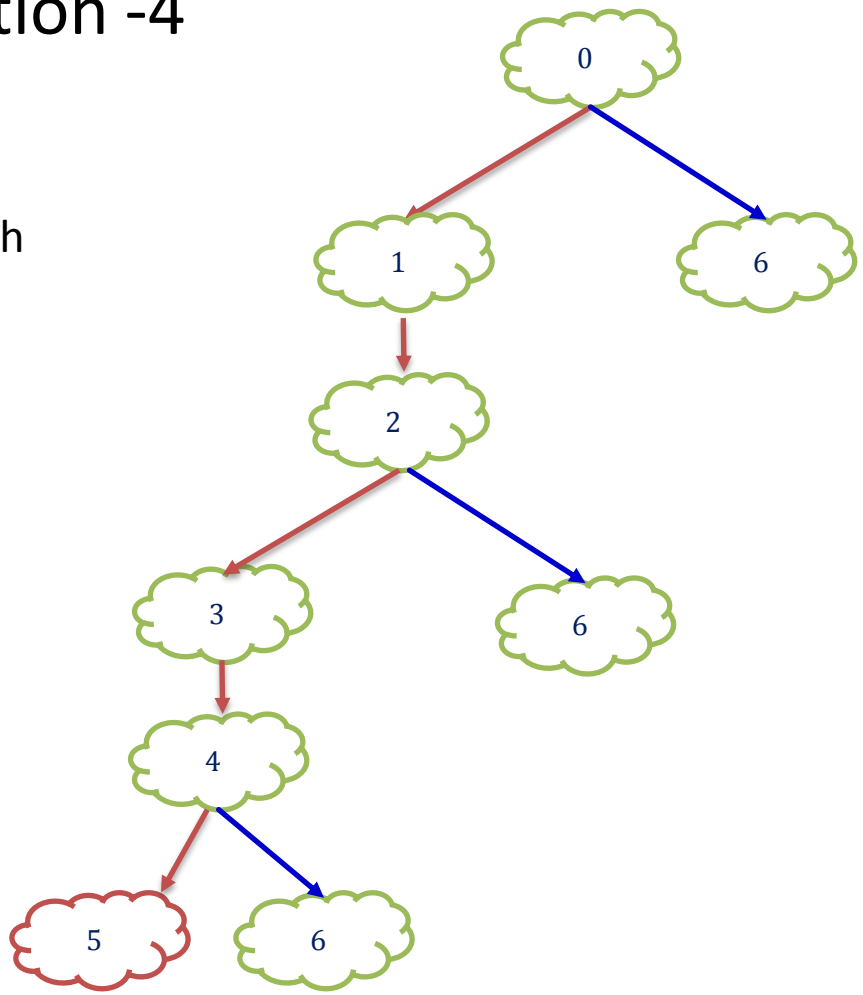
Step4:
Suitable test be a = 5.



Iteration -4

Step1: Got the witness and value of a which violates the safety property is 5.

Terminated …



Limitations

- Theorem prover is used to do test generation and maintaining abstraction after each refinement.
- Expensive to use theorem prover.
- Doesn’t handle pointers
- Doesn’t handle functions.

Idea !!

- Can we choose refinement technique so that theorem prover can be used only for test generation and maintaining abstraction will be a by product of failed test generation ??
  - Notion of Template based refinement.
- Can we handle function calls by recursively invocating algorithm by properly changing inputs and error condition ??

Idea !!

- Can we handle pointers by doing whole program static pointer aliasing ?
    - Lot of over approximation !
- Can we do by flow sensitive pointer aliasing ?
    - Finding possible aliases in the error path up to frontier.
- How can we incorporate this analysis to refine ??
    - Combine with Weakest pre condition operator !

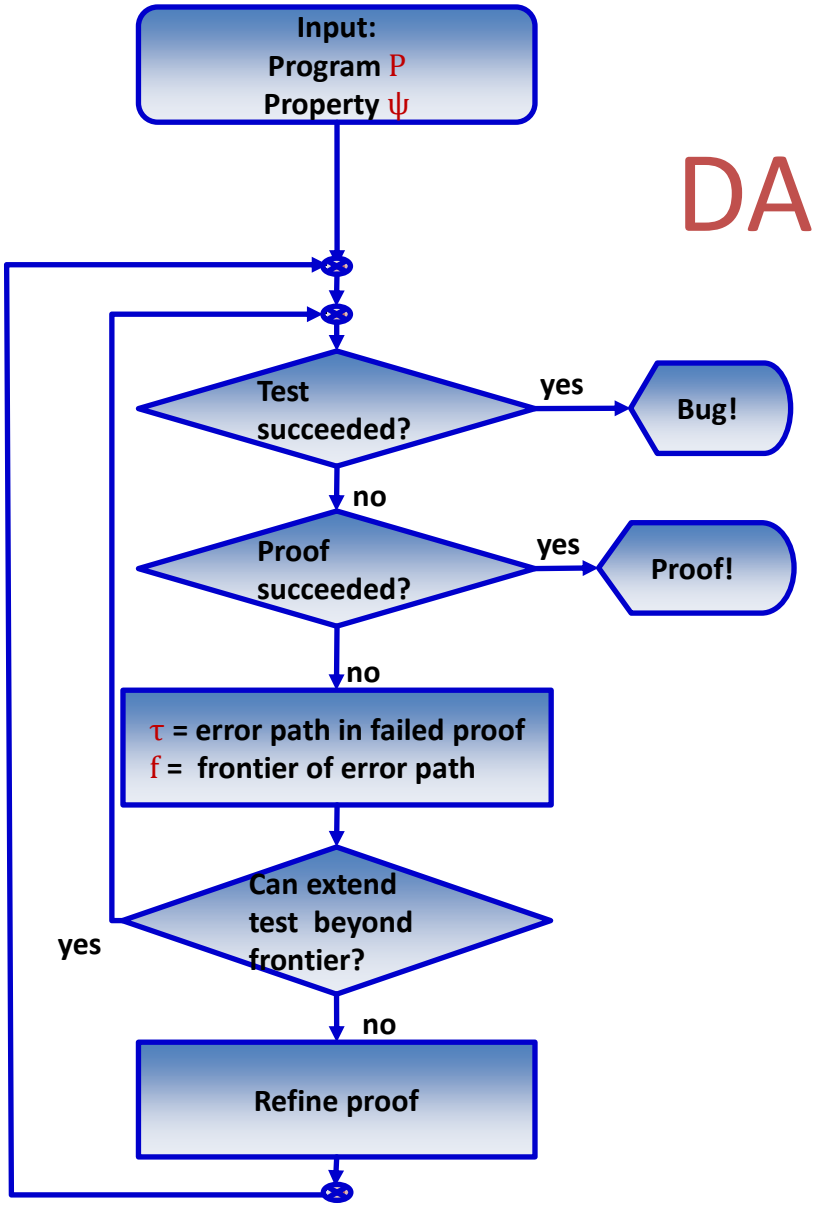
Outline

- Overview of Symbolic execution [DONE]
- Overview of (DART) [DONE]
- Synergy and its working on examples [DONE]
- DASH and its working on examples
- SMASH and its working on examples
- Problem statement and Proposed solution
- Conclusion

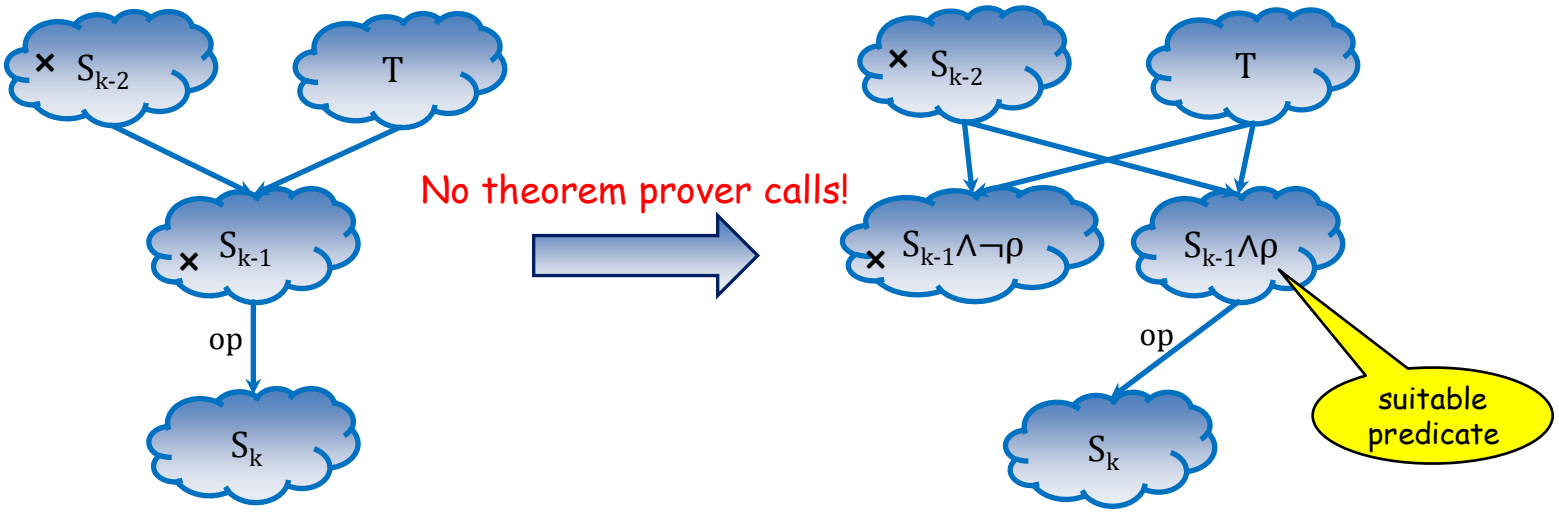
Overview of DASH

Extension of SYNERGY by adding above ideas

DASH Sketch


Input:
Program P
Property ψ
Test succeeded?
yes
Bug!
no
Proof succeeded?
yes
Proof!
no
τ = error path in failed proof
f = frontier of error path
Can extend test beyond frontier?
yes
no
Refine proof

Refinement using Suitable predicates



Here theorem prover is not used for maintaining abstraction.

- Weakest pre condition operator:
  - Explodes in size of constraints in presence of pointers
  - Ex: {pre} { i = j} { *a < 10 }
    - Possible combinations are "a=&i ^ j < 10" and
    - "a != &i ^ *a < 10}
  - {pre} { i = j } { *a + *b < 10 }
    - 4 possible combinations

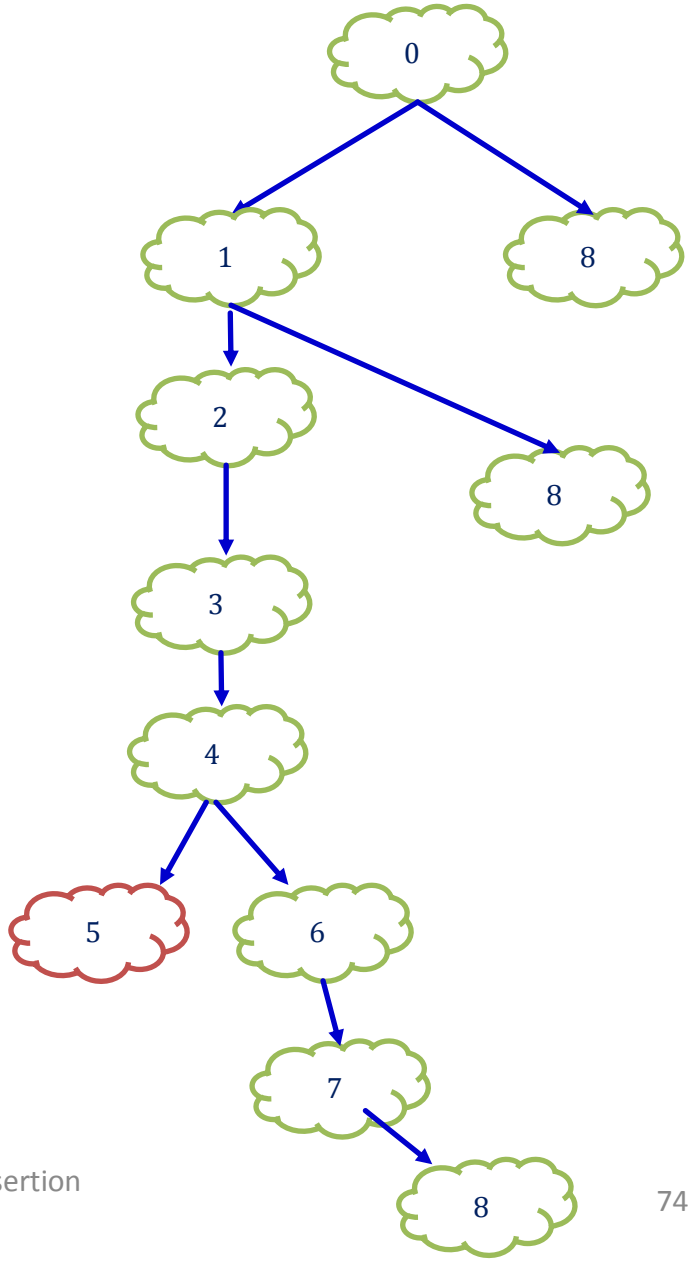
- Notion of Weakest pre condition alpha operator :
  - Choose alias combination obtained only in the current path.
  - This operator is stronger than weakest pre condition.

Handling Pointers

$$\rho = WP \downarrow (op, \psi) = \alpha \wedge WP(op, \psi)$$
$$WP_{\alpha}(op, \psi) = \neg(\alpha \vee WP \downarrow (op, \psi))$$

Example containing Pointers

```
void alias(int *p, int *p1, int *p2)
{
0: if (p == p1) return;
1: if (p == p2) return;
2: *p1 = 0; *p2 = 0;
3: *p = 1;
4: if (*p1 == 1 || *p2 == 1)
5:             error();
6: p = p1;
7: p = p2;
8: }
```

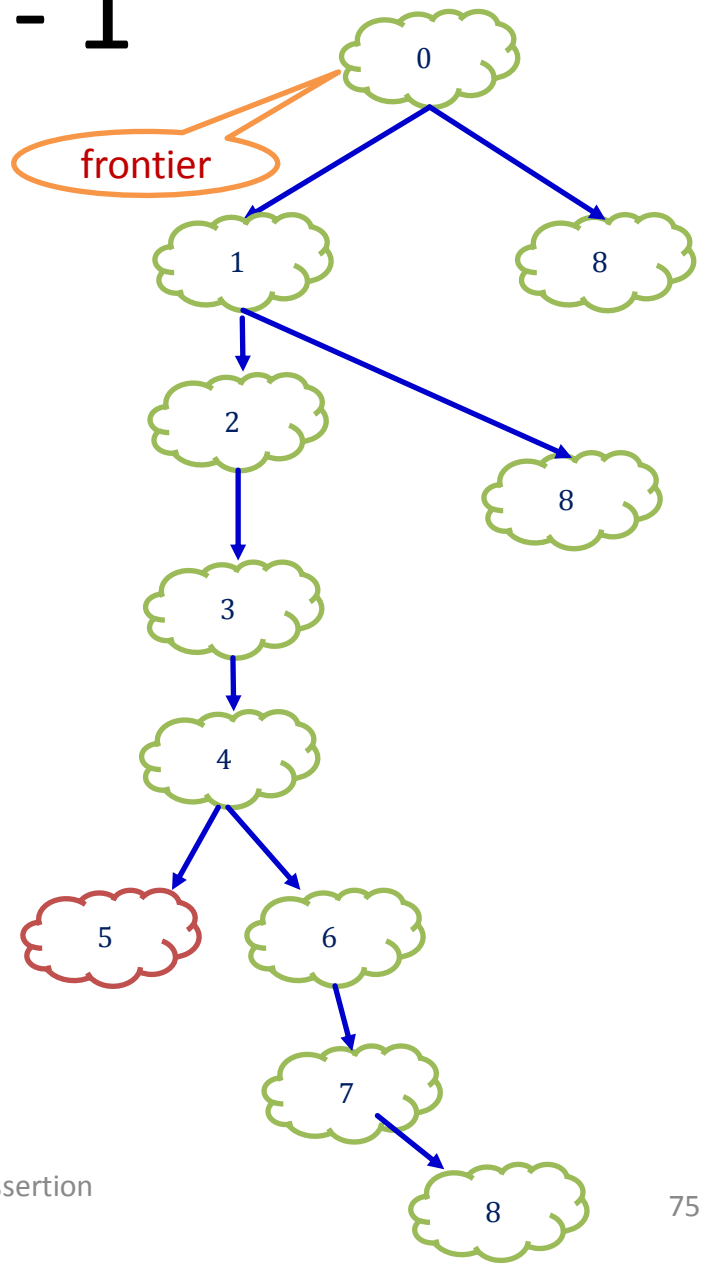


Iteration - 1

Step1: No witness

Step2: No proof

Step3: Abstract error trace is {0,1,2,3,4,5} and
Frontier is 0.



Iteration - 1

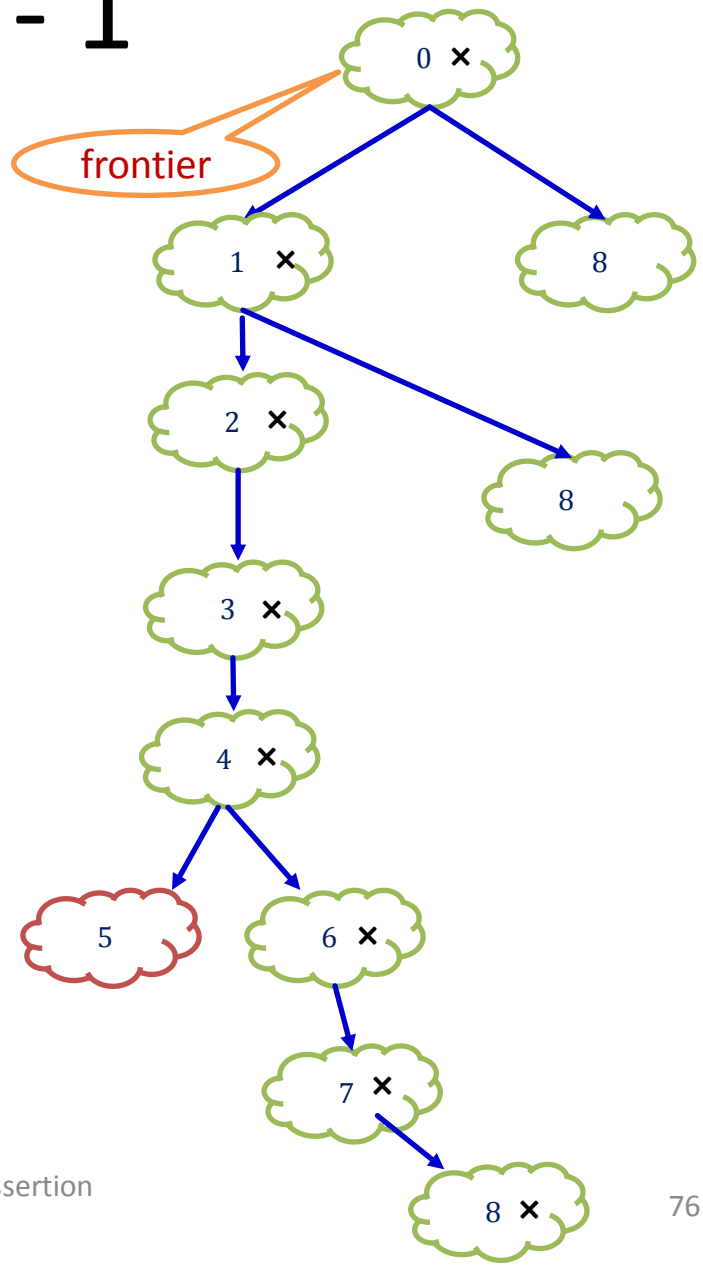
Step1: No witness

Step2: No proof

Step3: Abstract error trace is {0,1,2,3,4,5} and
Frontier is 0.

Step4: Random generation of inputs p,p1 and p2 pointing to different locations.

Forest gets updated by a concrete path from 0 to 8.



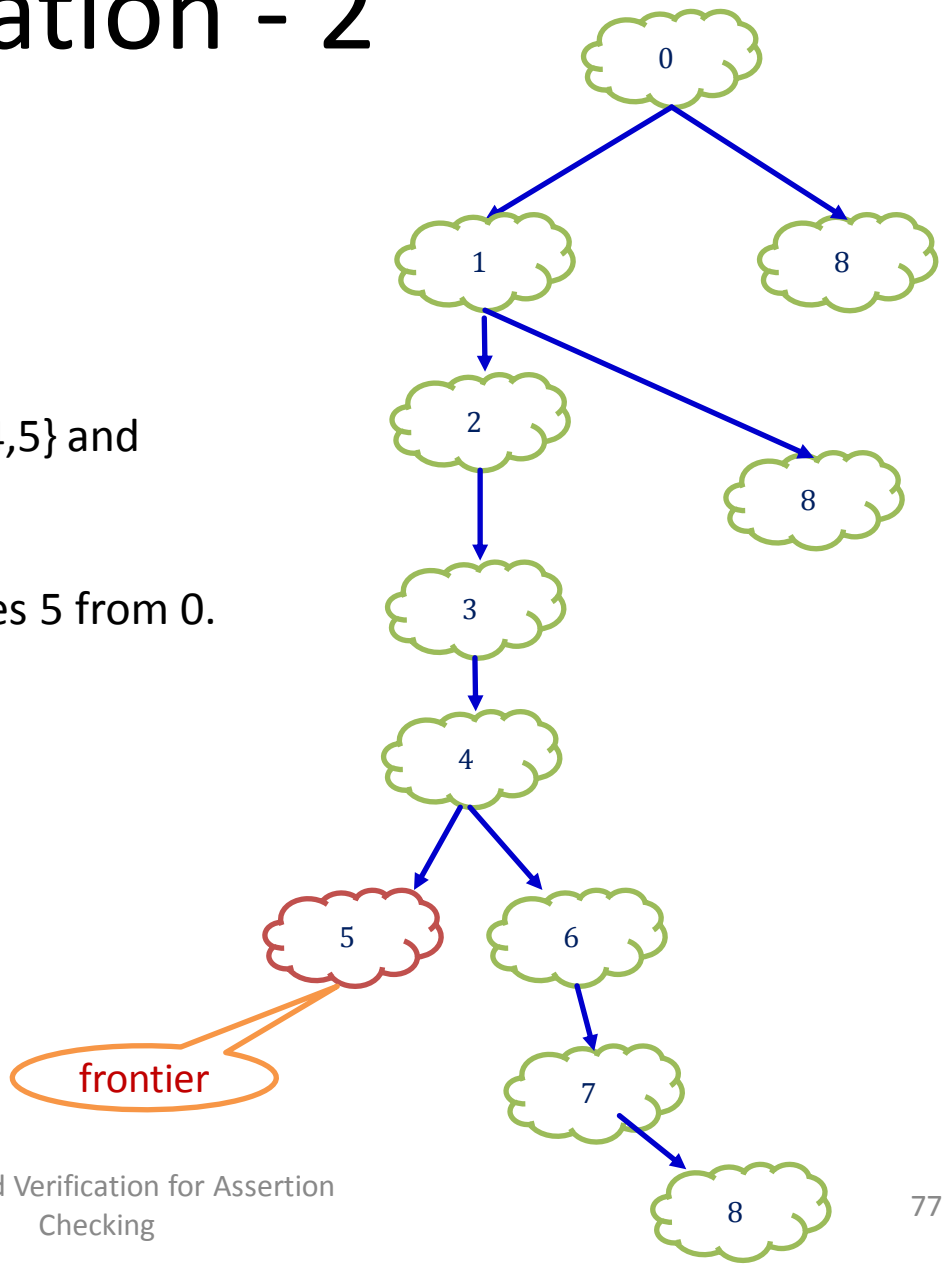
Iteration - 2

Step1: No witness

Step2: No proof

Step3: Abstract error trace is {0,1,2,3,4,5} and
Frontier is 5.

Couldn't generate an input that reaches 5 from 0.



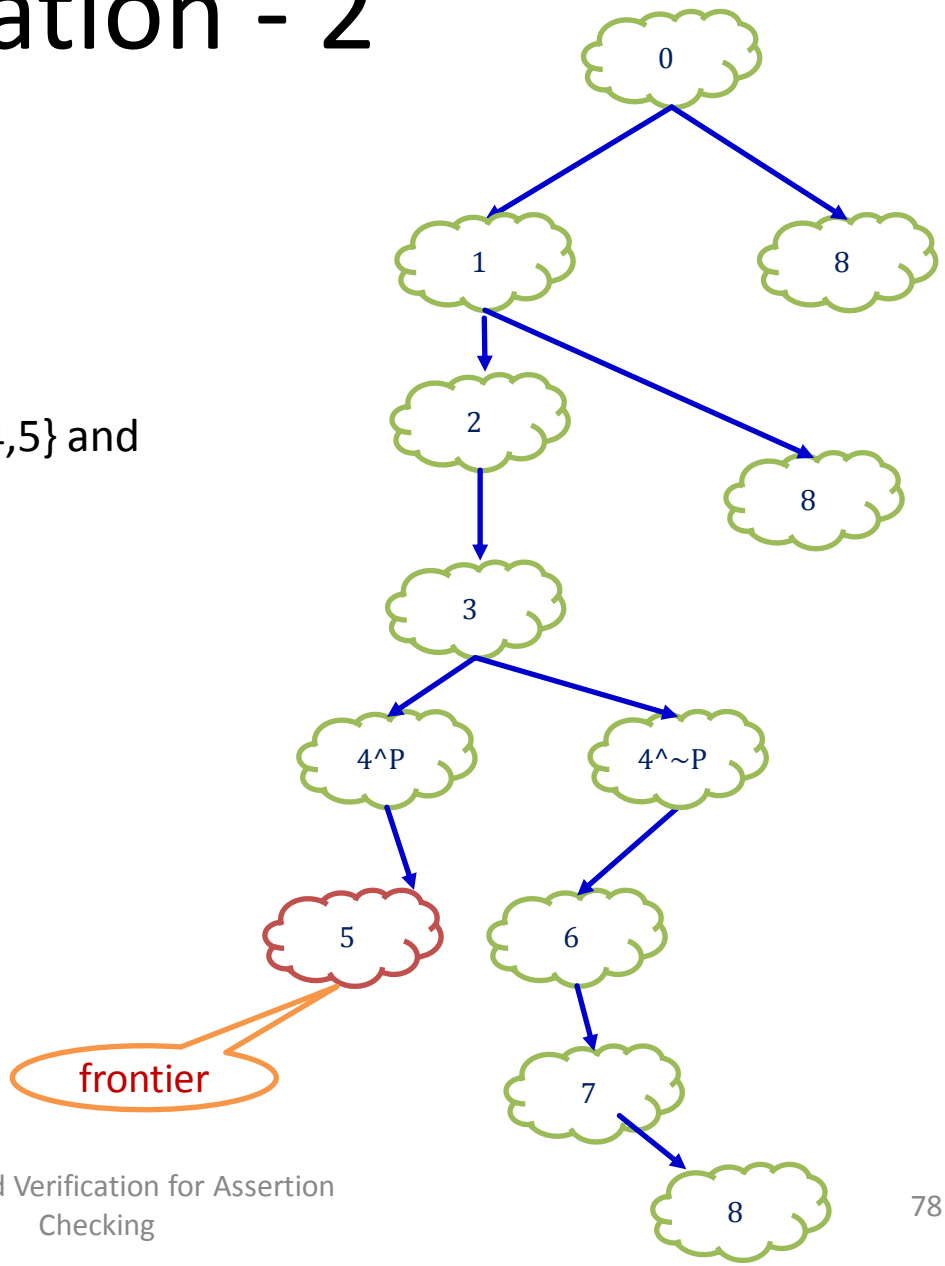
Iteration - 2

Step1: No witness

Step2: No proof

Step3: Abstract error trace is {0,1,2,3,4,5} and
Frontier is 5.

Refining 4 into 4^P and 4^~P where
P = (*p1 == 1 || *p2 == 1)



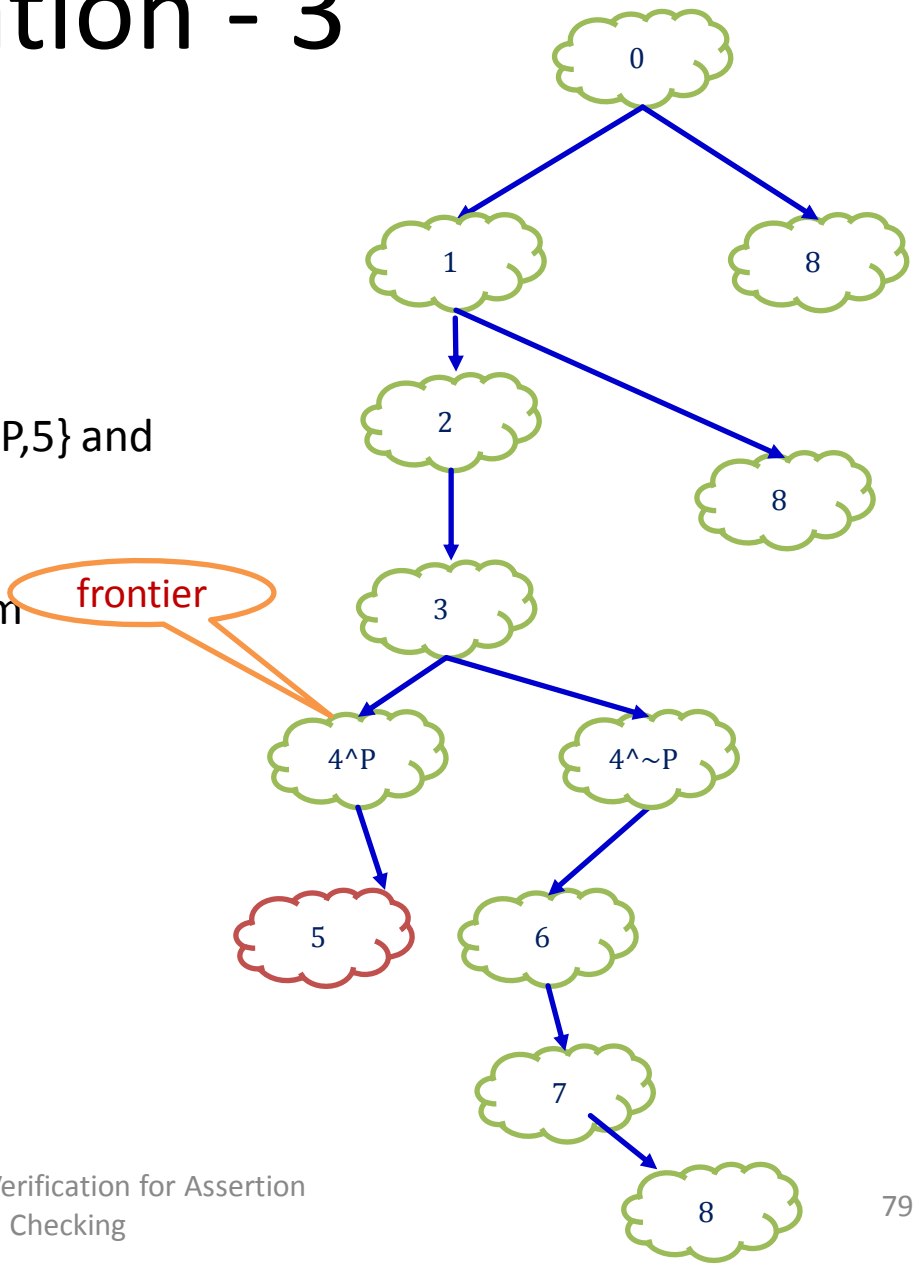
Iteration - 3

Step1: No witness

Step2: No proof

Step3: Abstract error trace is {0,1,2,3,4^P,5} and
Frontier is 4^P.

Couldn't generate an input to move from 4^P to 5.



Iteration - 3

Step1: No witness

Step2: No proof

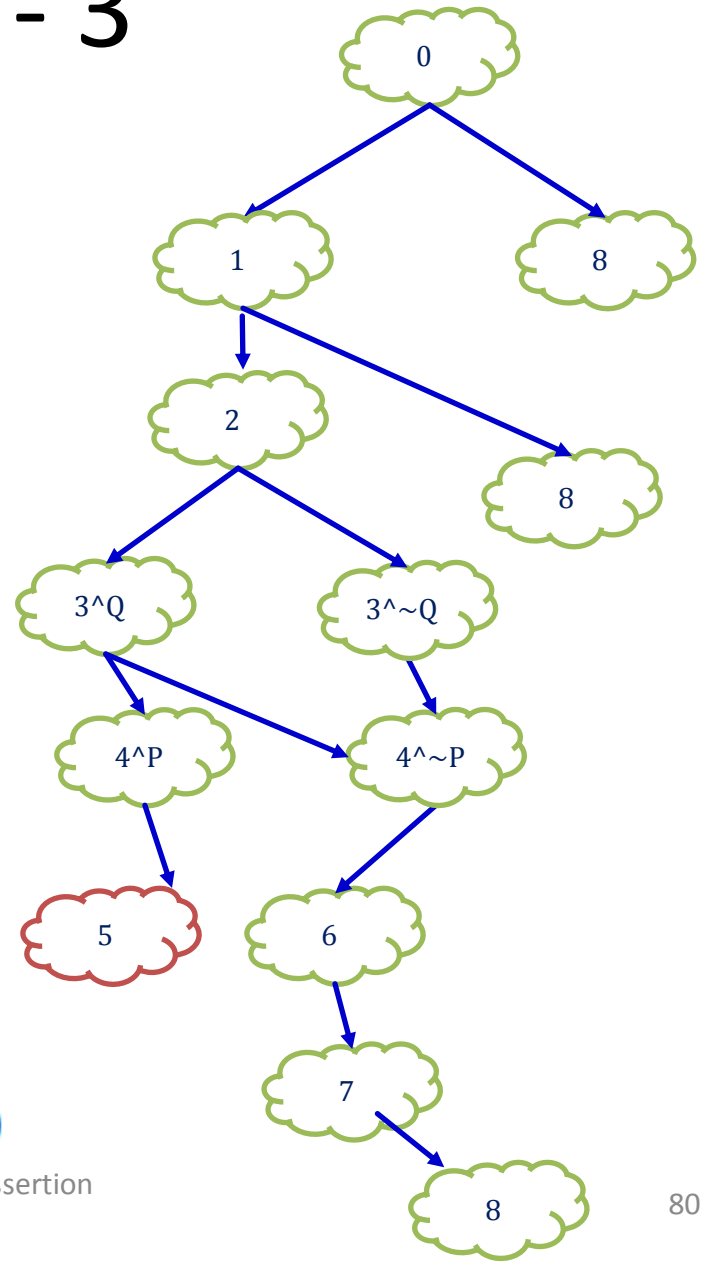
Step3: Abstract error trace is {0,1,2,3,4^P,5} and
Frontier is 4^P.

Alias combination: { p != p1 and p != p2}

Refining 3 into 3^Q and 3^~Q where
α = (p != p1 || p != p2)
WP|α = = (*p1 == 1 || *p2 == 1)

Q = ~(α ^ ~WP|α)
= (p == p1 & p == p2) || *p1 = 1 || *p2 =1

$$\mathrm{WP}_{\alpha}(op, \phi_2) \stackrel{\text{def}}{=} \neg(\alpha \wedge \neg \mathrm{WP}\downarrow_{\alpha}(op, \phi_2))$$



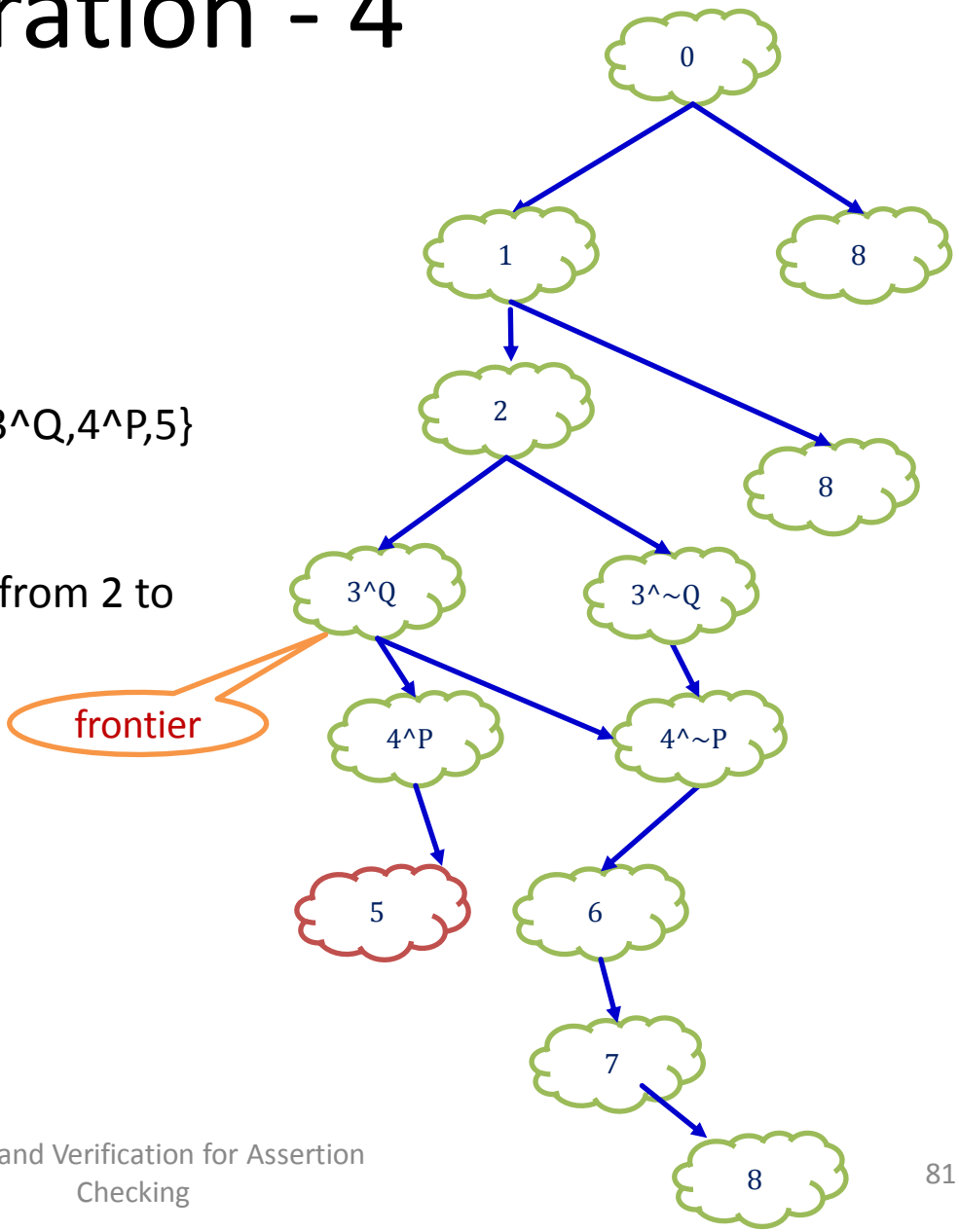
Iteration - 4

Step1: No witness

Step2: No proof

Step3: Abstract error trace is {0,1,2,3^Q,4^P,5} and Frontier is 3^Q.

Couldn’t generate an input to move from 2 to 3^Q.



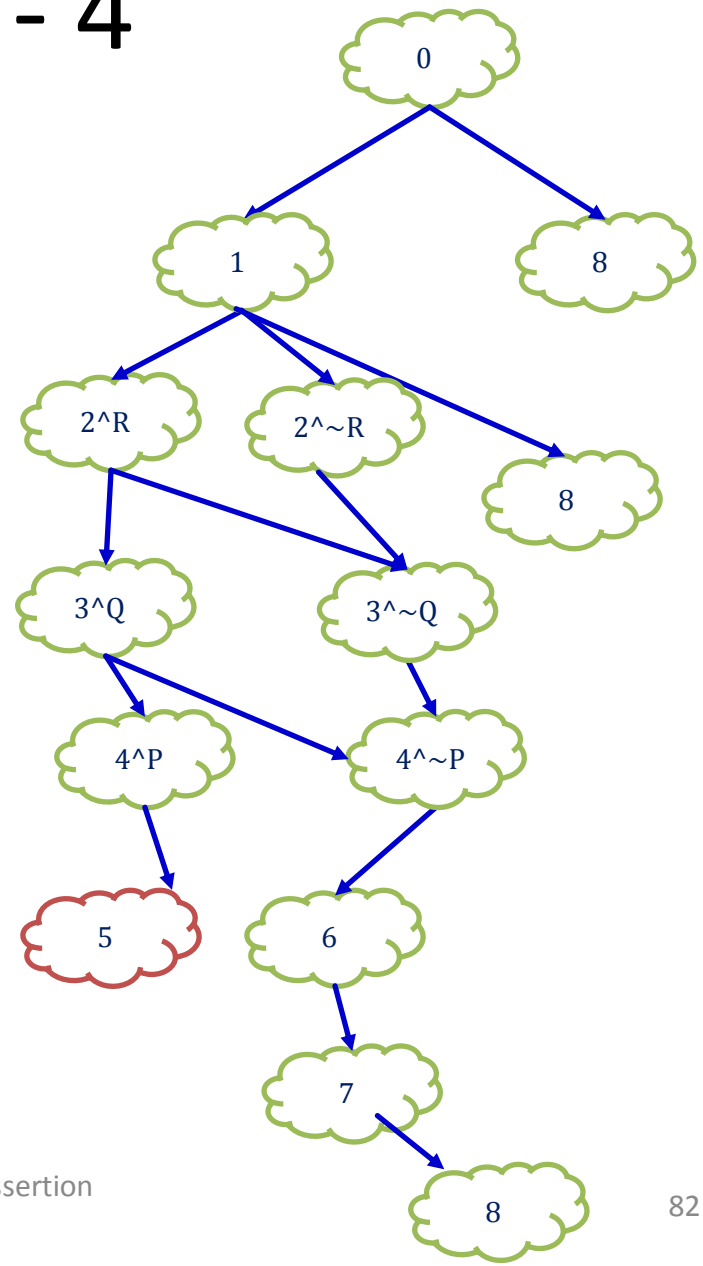
Iteration - 4

Step1: No witness

Step2: No proof

Step3: Abstract error trace is {0,1,2,3^Q,4^P,5} and Frontier is 3^Q.

Refining 2 into 2^R and 2^~R where
R = (p == p1 & p == p2)



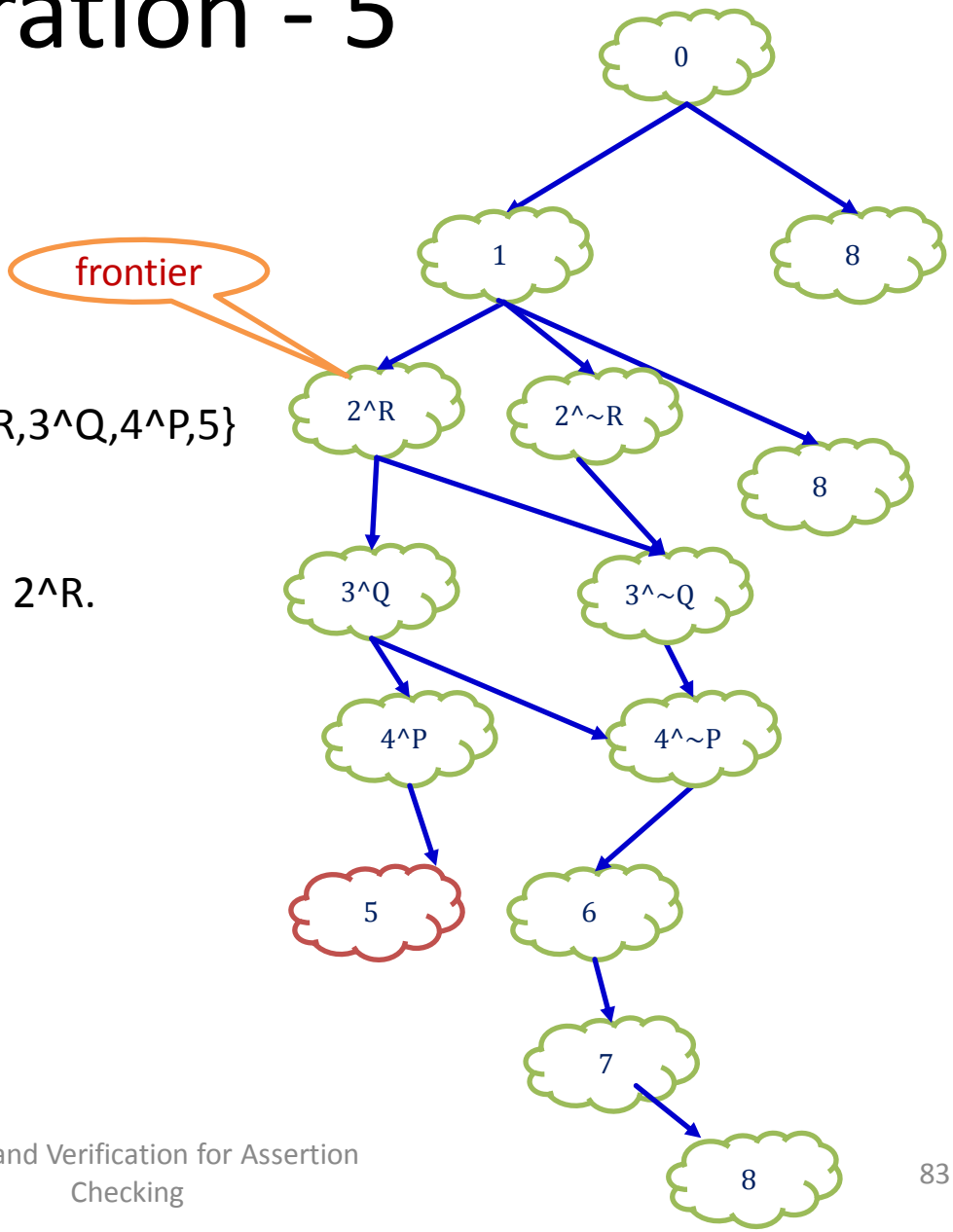
Iteration - 5

Step1: No witness

Step2: No proof

Step3: Abstract error trace is {0,1,2^R,3^Q,4^P,5} and Frontier is 2^R.

Couldn't generate an input from 1 to 2^R.



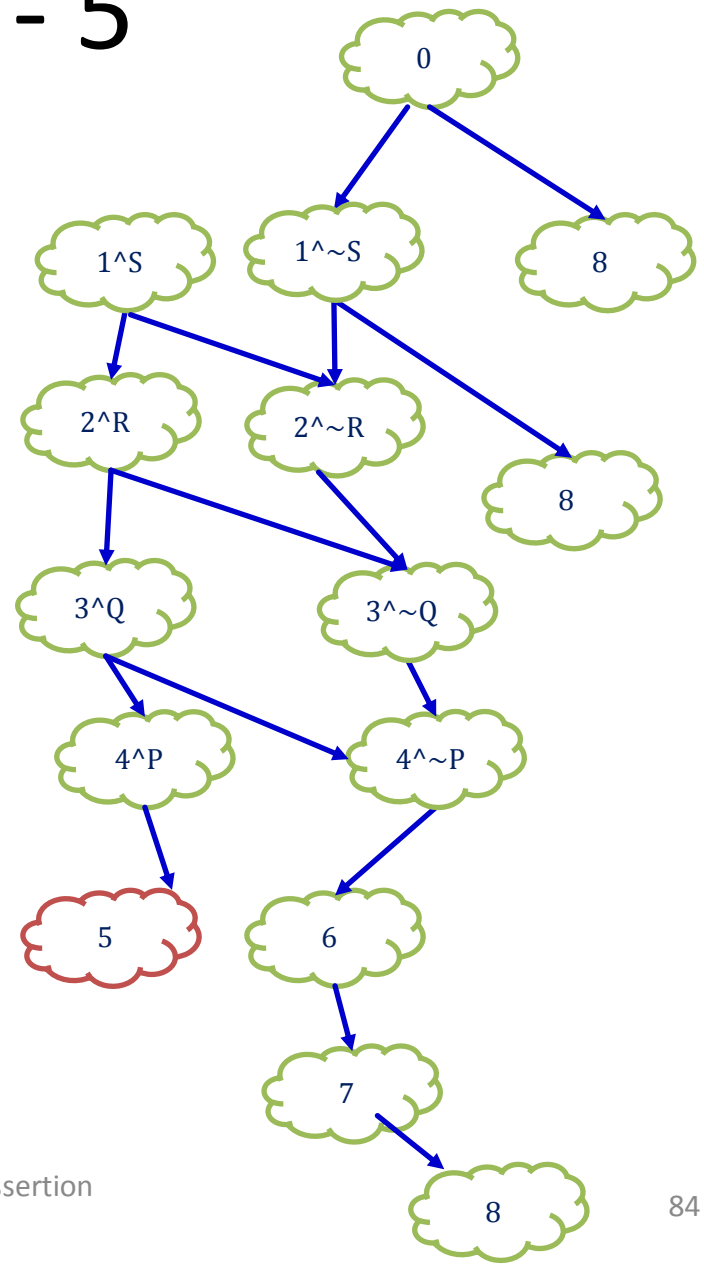
Iteration - 5

Step1: No witness

Step2: No proof

Step3: Abstract error trace is {0,1,2^R,3^Q,4^P,5} and Frontier is 2^R.

Splitting 1 into 1^S and 1^~S where
WPα (if(p != p2), R) = false = S

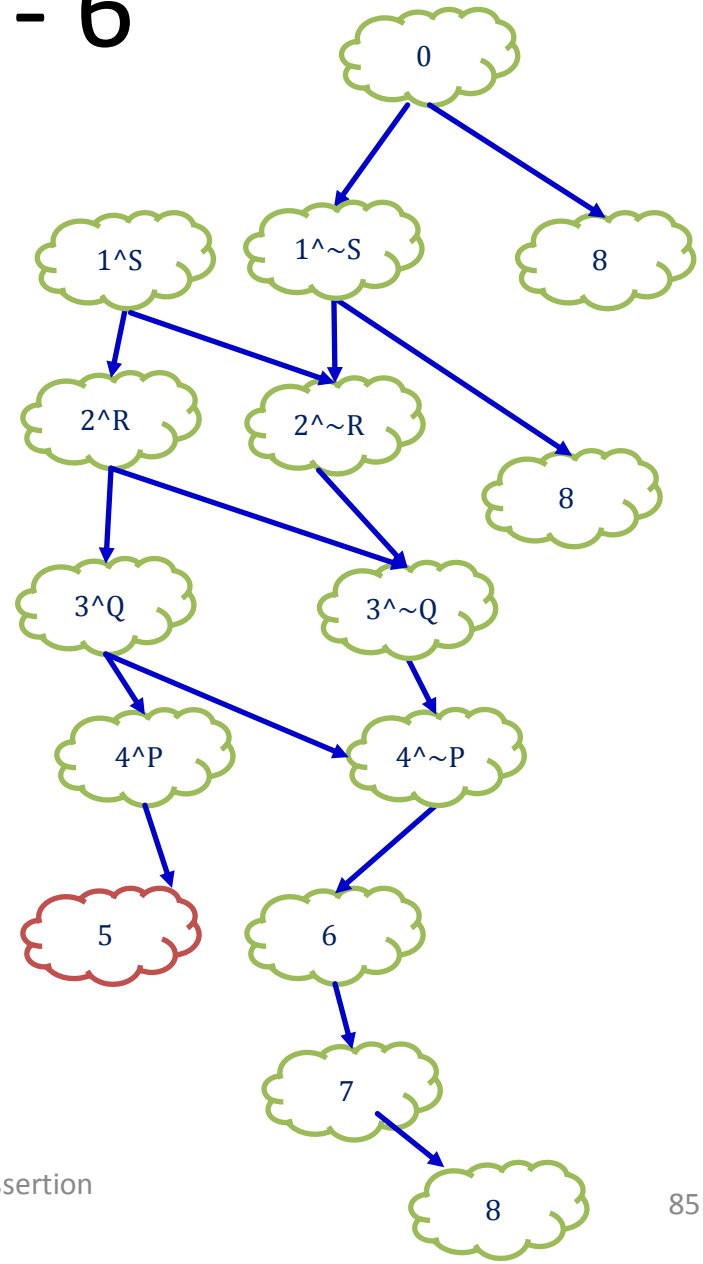


Iteration - 6

Step1: No witness

Step2: But proof

If there are n pointers and if we use only weakest pre condition operator, then we have exponential blow in number of constraints.



Example containing Functions

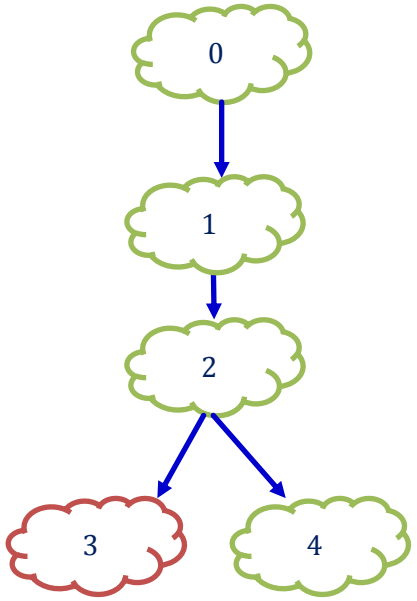
Separate Abstraction and Forest for each function call

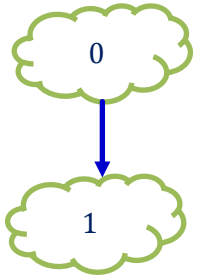
Note points:

1. Handling when frontier edge is a function call

2. Handling when function calls are in between initial node to node before frontier edge.

```
void top(int x) {
            int a,b;
0:          a = inc(x);
1:          b = inc(a);
2:          if ( b!= x + 2)
3:                      error;
4:          return;
}

int inc (int y) {
            int r;
0:          r = y+1;
1:          return r;
}
```



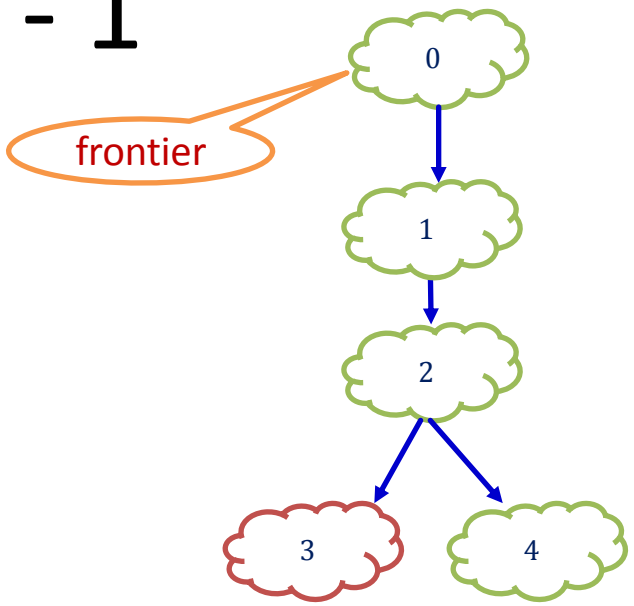


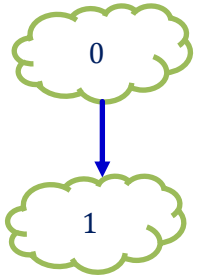
Iteration - 1

Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1, 2,3} and Frontier is 0





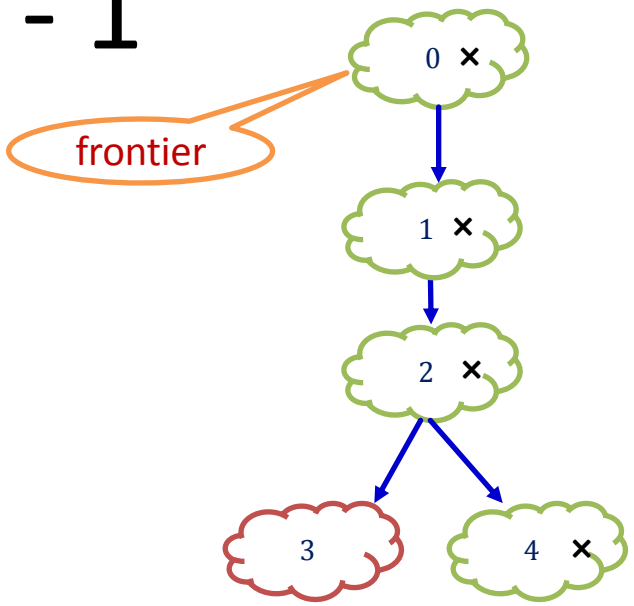
Iteration - 1

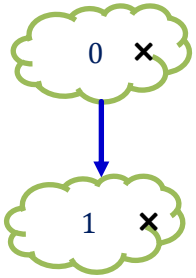
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1, 2,3} and Frontier is 0

Could move from 0 to 1 by generating a test case as x = 10.



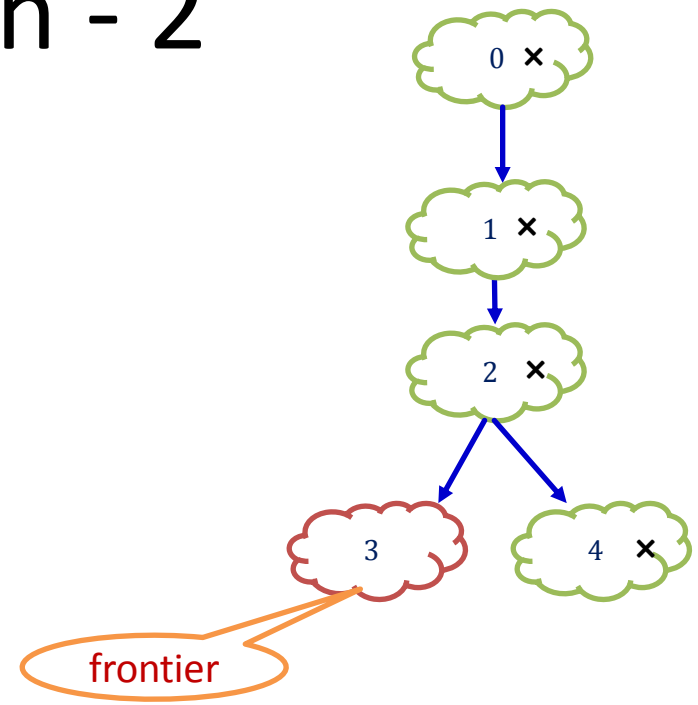


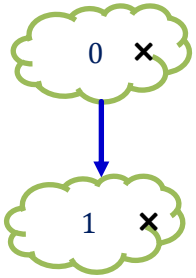
Iteration - 2

Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1, 2,3} and Frontier is 3





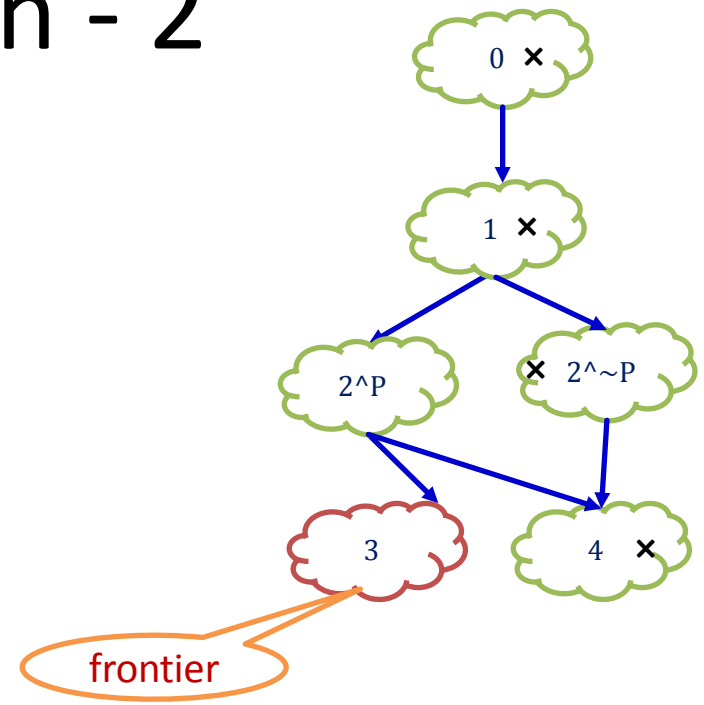
Iteration - 2

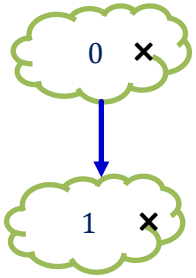
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1, 2,3} and Frontier is 3

Couldn't generate a test case from 2 to 3. So, refining 2 into 2^P and 2^~P where P = (b!= x+2)





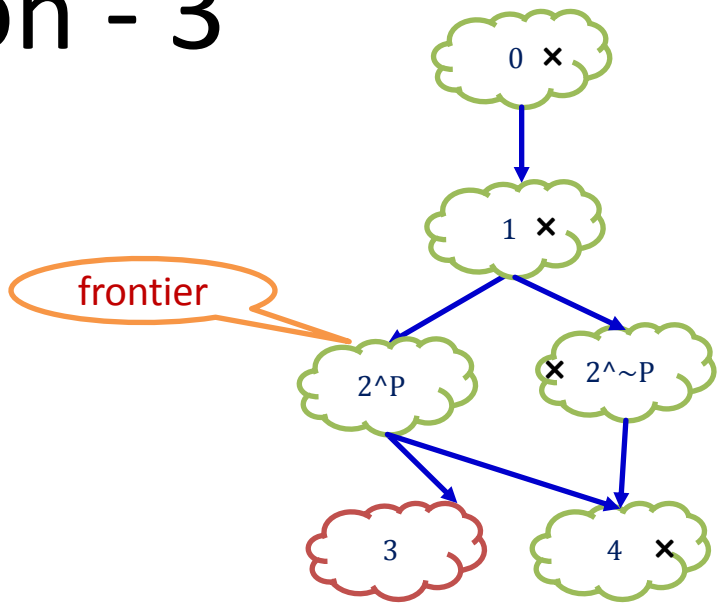
Iteration - 3

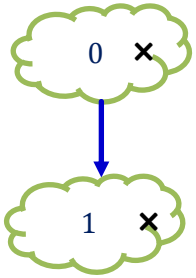
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1, 2^P,3} and Frontier is 2^P

Try to extend 1 to 2^P.
Interprocedural call
Pre condition: (a = x + 1) and y = a
Post condition: ret != x + 2
DASH(inc, y = x + 1, ret != x + 2);





Iteration - 3

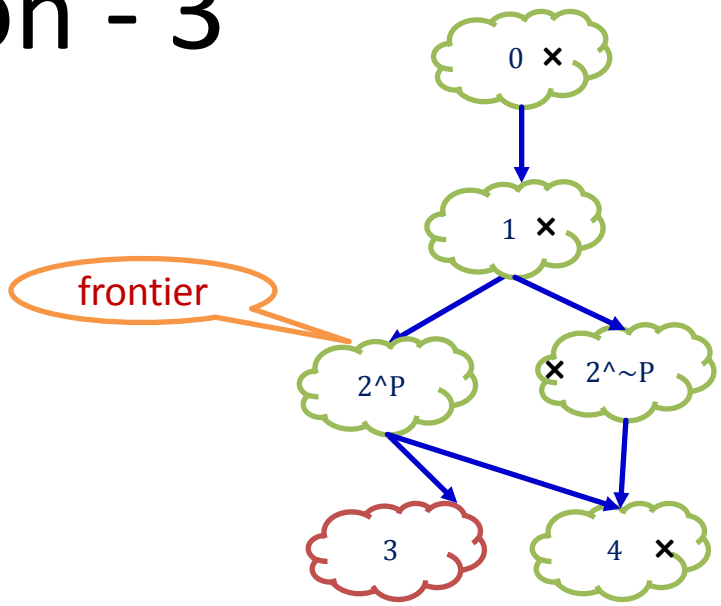
Recursive call – 1 – Iteration - 1

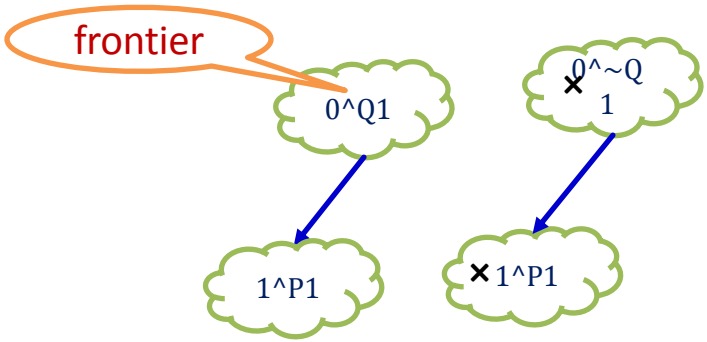
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is {0,1^P1} and
Frontier is 1^P1

Couldn't generate an input and refining 0 into
0^Q1 and 0^~Q1
Where Q1 = (y != x + 1}
Q1 {ret = y } { ret != x + 2}
0^Q1 = false





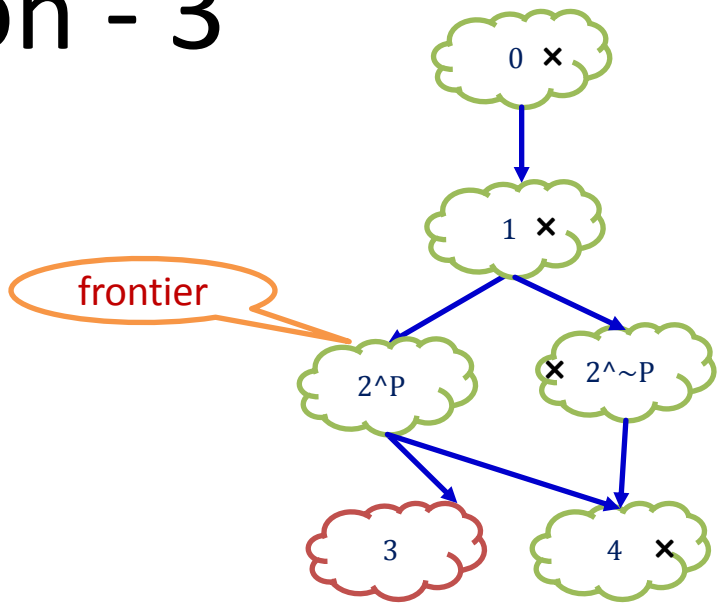
Iteration - 3

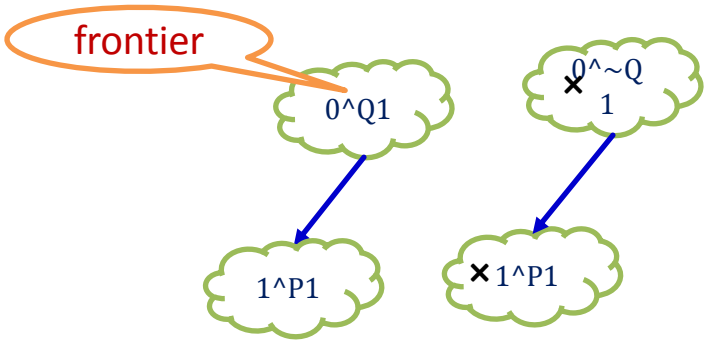
Recursive call – 1 – Iteration - 2

Step1: Couldn't get witness

Step2: Could get proof

Predicate used to partition is
y != x + 1





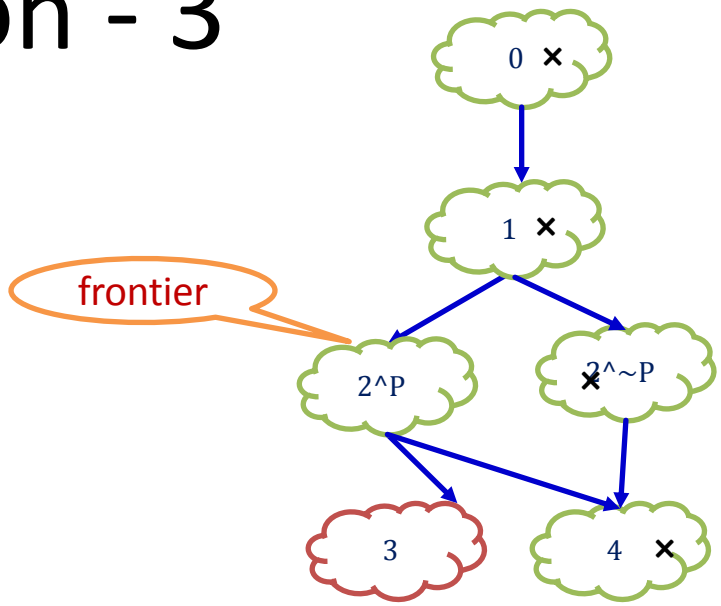
Iteration - 3

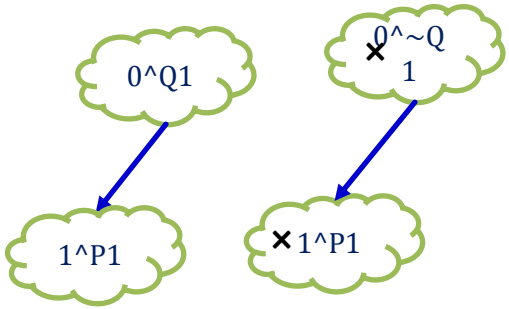
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1, 2^P,3} and Frontier is 2^P

Try to extend 1 to 2^P.
Interprocedural call
Pre condition: (a = x + 1) and y = a
Post condition: ret != x + 2
DASH(inc, y = x + 1, ret != x + 2);
Predicate to refine: y != x + 1





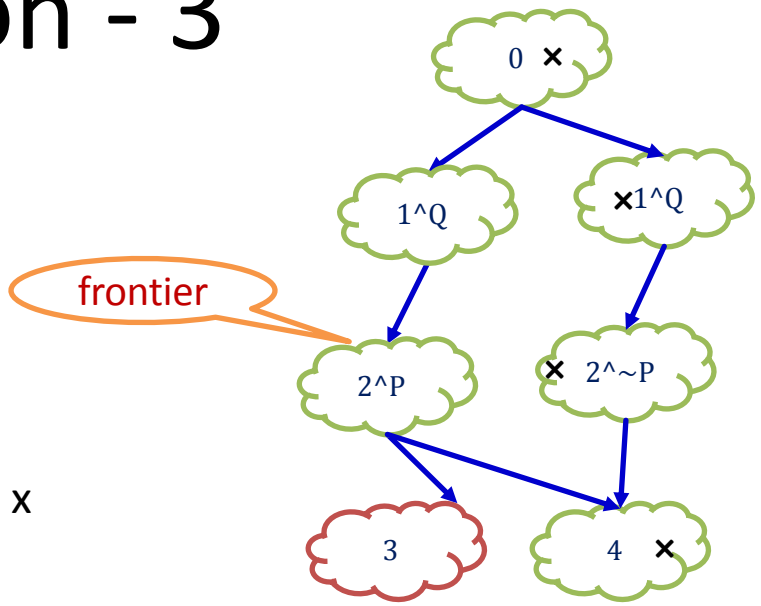
Iteration - 3

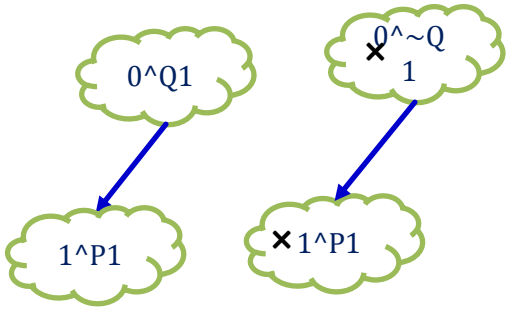
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1, 2^P,3} and Frontier is 2^P

Splitting 1 into 1^Q and 1^~Q where Q = (a != x + 1)





Iteration - 4

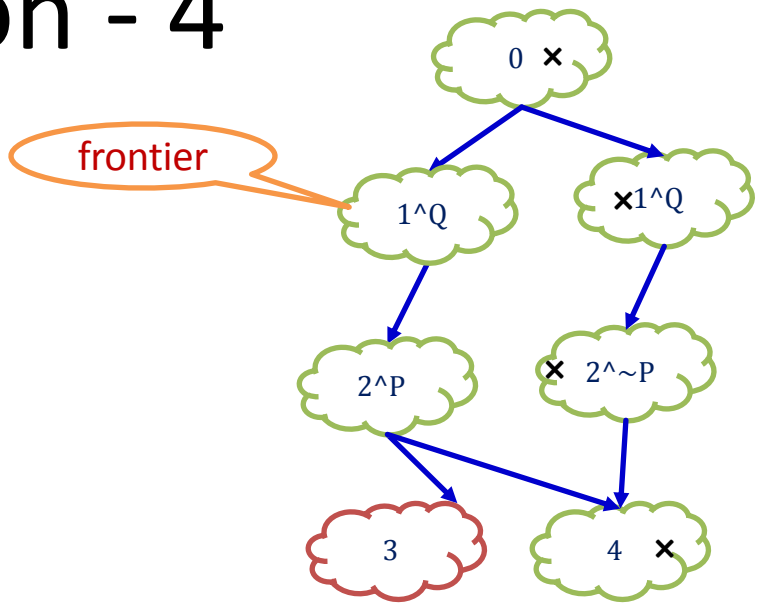
Step1: Couldn't get witness

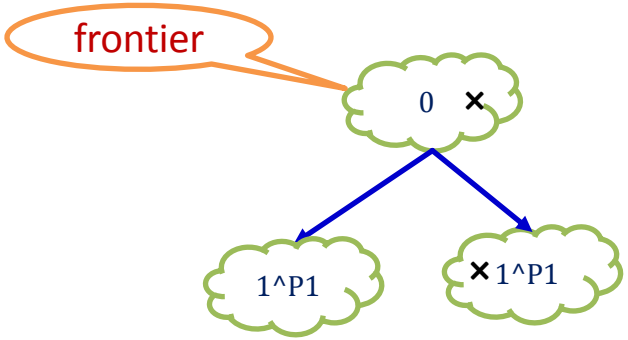
Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1^Q, 2^P,3} and Frontier is 1^Q

Interprocedural call:
Pre condition: true
Post condition: a != x + 1

DASH (inc, true, a != x + 1);
P1 = (ret != x + 1)





Iteration - 4

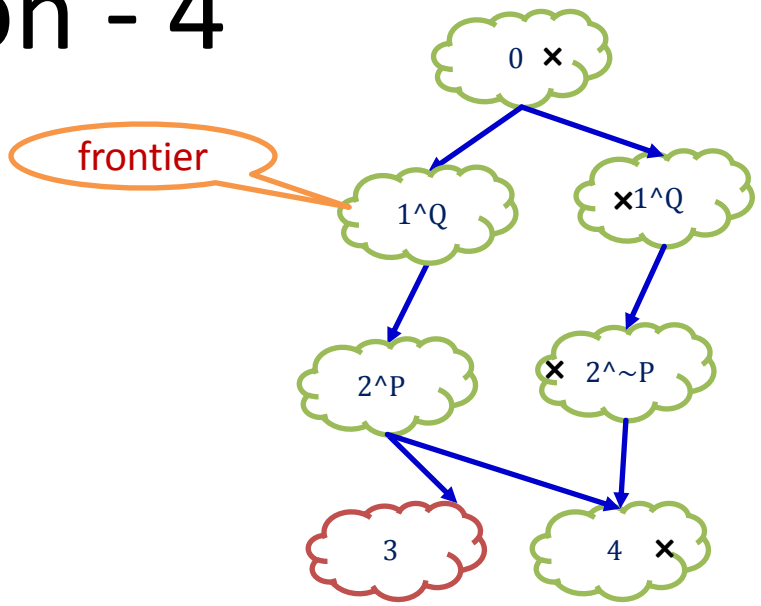
Recursive call – 2 Iteration - 1
Step1: Couldn't get witness

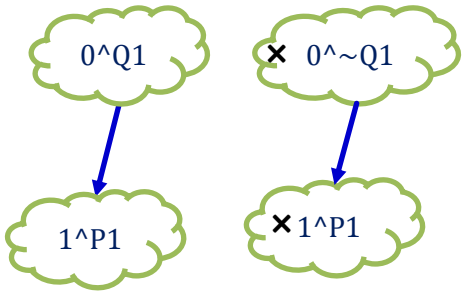
Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1^P1} and
Frontier is 0

Couldn't generate an input from 0 to 1^P1.

Refining 0 into O^Q1 and 0^~Q1
Where Q1 = false
Q1 { ret = y + 1} { ret != y + 1}

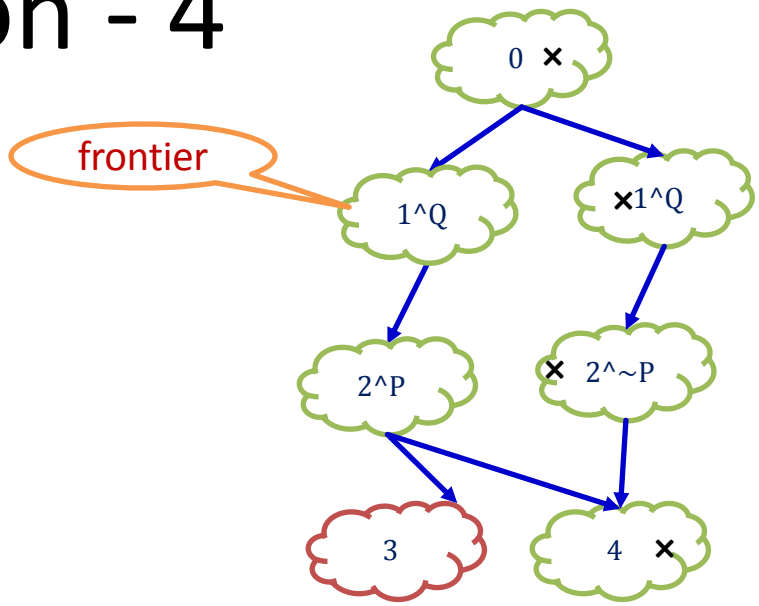


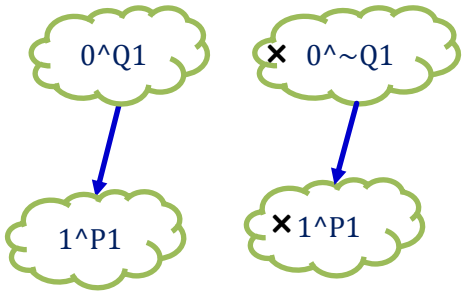


Iteration - 4

Recursive call – 2 Iteration - 1
Step1: Couldn't get witness

Step2: Could get proof





Iteration - 4

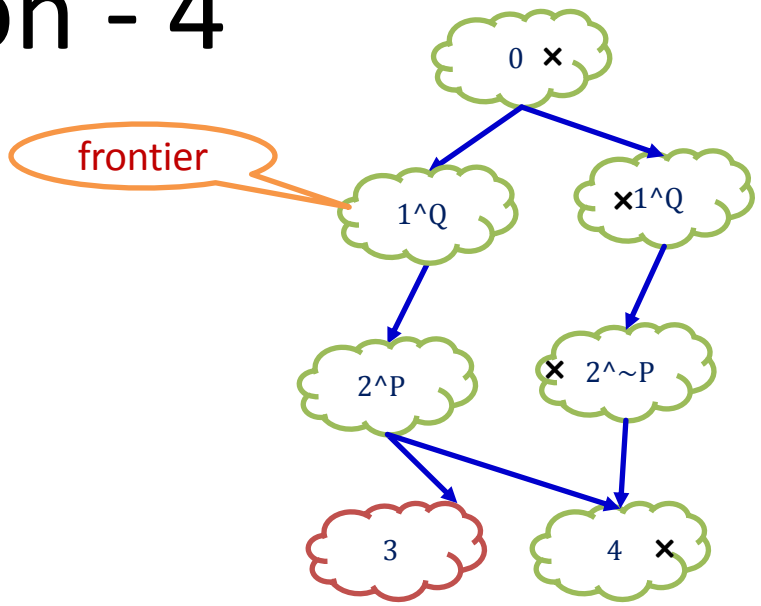
Step1: Couldn't get witness

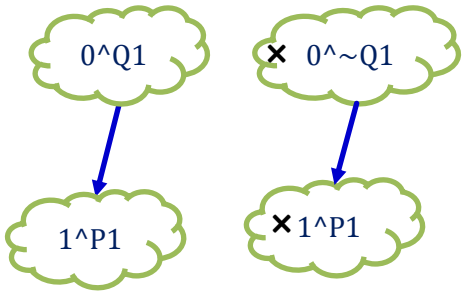
Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1^Q, 2^P,3}
and Frontier is 1^Q

Interprocedural call:
Pre condition: true
Post condition: a != x + 1

DASH (inc, true, a != x + 1);
P1 = (ret != x + 1)
Predicate used to split: false





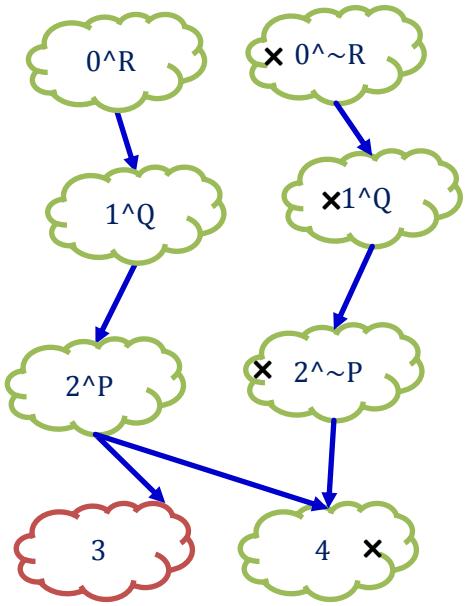
Iteration - 4

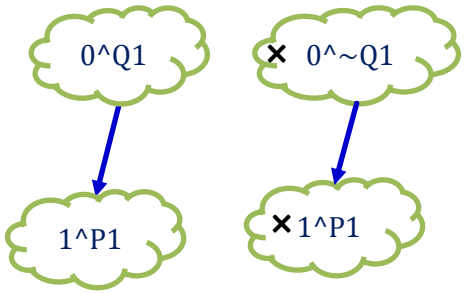
Step1: Couldn't get witness

Step2: Couldn't get proof

Step3: Abstract error trace is {0, 1^Q, 2^P,3}
and Frontier is 1^Q

Refining 0 to 0^R and O^~R where
R = false

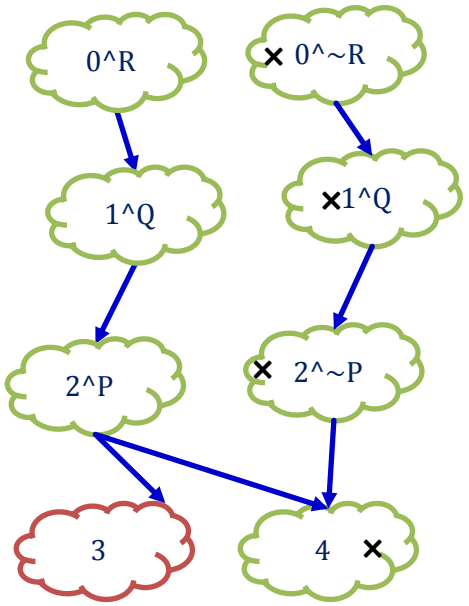


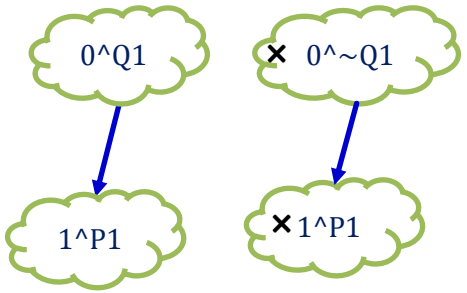


Iteration - 5

Step1: Couldn't get witness

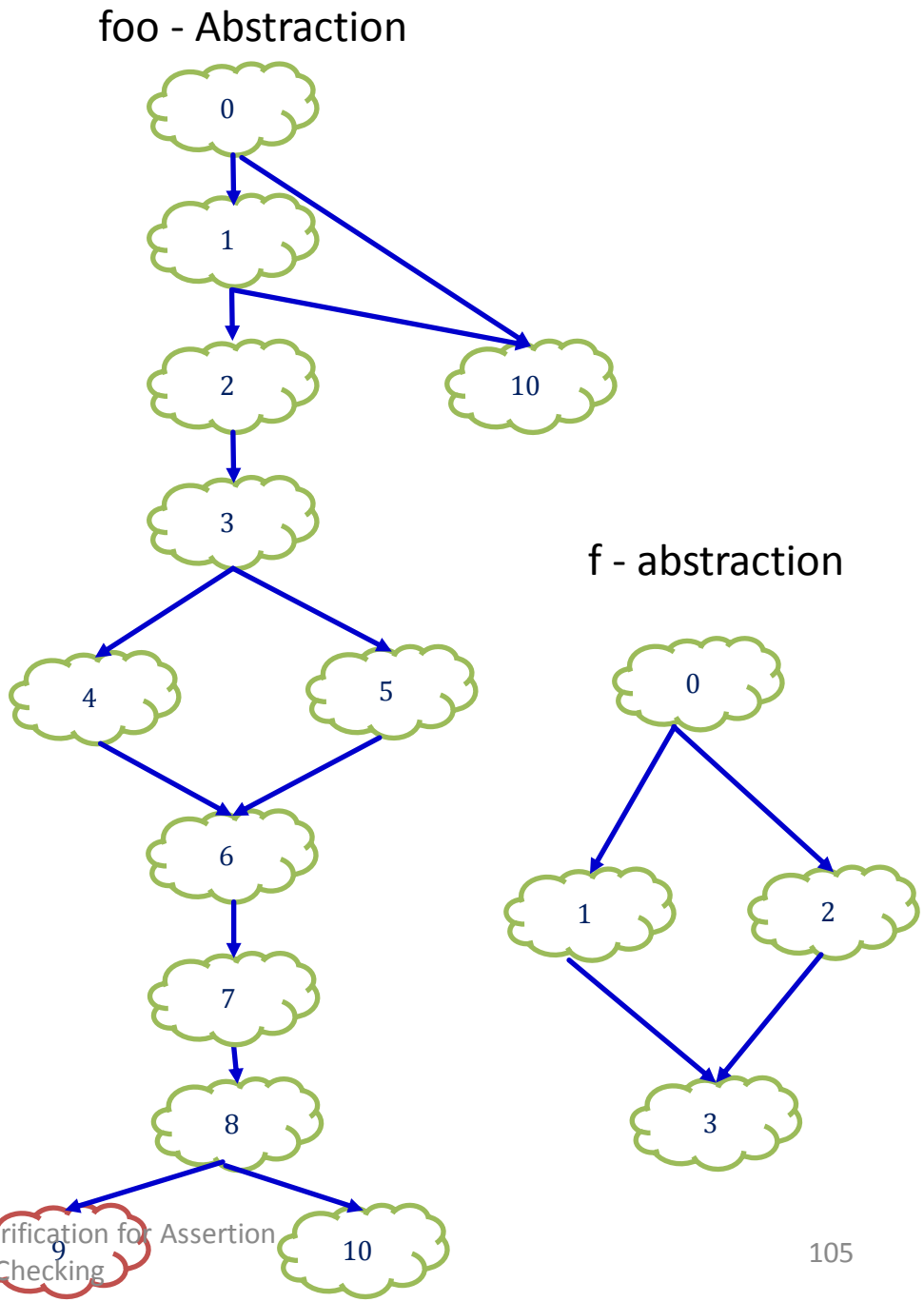
Step2: Could get proof





Example on both pointers and Functions

```
   void  foo (int *x, int *y, int*z, int a ) {
0:    if ( x == y )
                return ;
1:    if ( x == z)
                return;
2:    int ret = f (a);
3:    if ( ret < 0 )
4:              x = z;
      else
5:              x = y;
6:    *y = 1
7:    *z = 2
8:    if ( *x + *y + *z == 5 )
9:              error ();
10: }

   int  f ( int a ) {
0:   if ( a < 0)
1:              return –a ;
     else
2:              return a ;
3: }
```

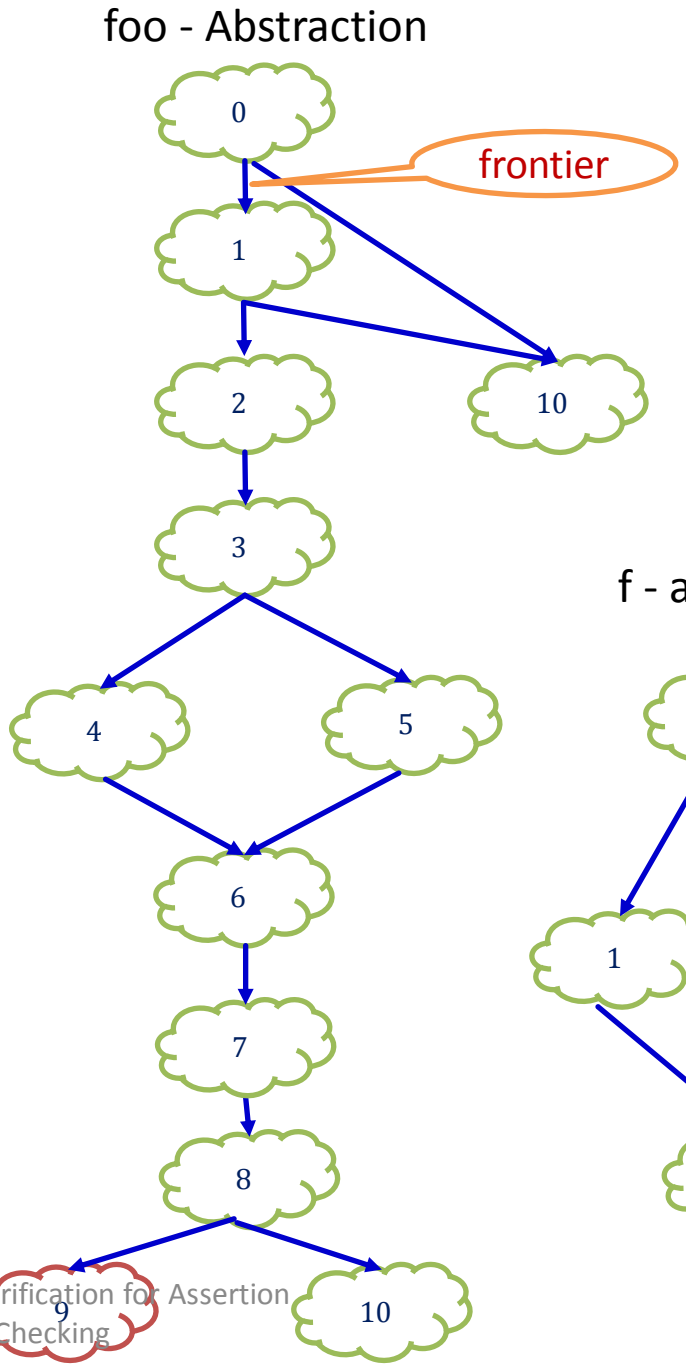


Iteration – 1

Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3, 4, 6, 7, 8, 9> and
Frontier is <0-1>



Iteration – 1

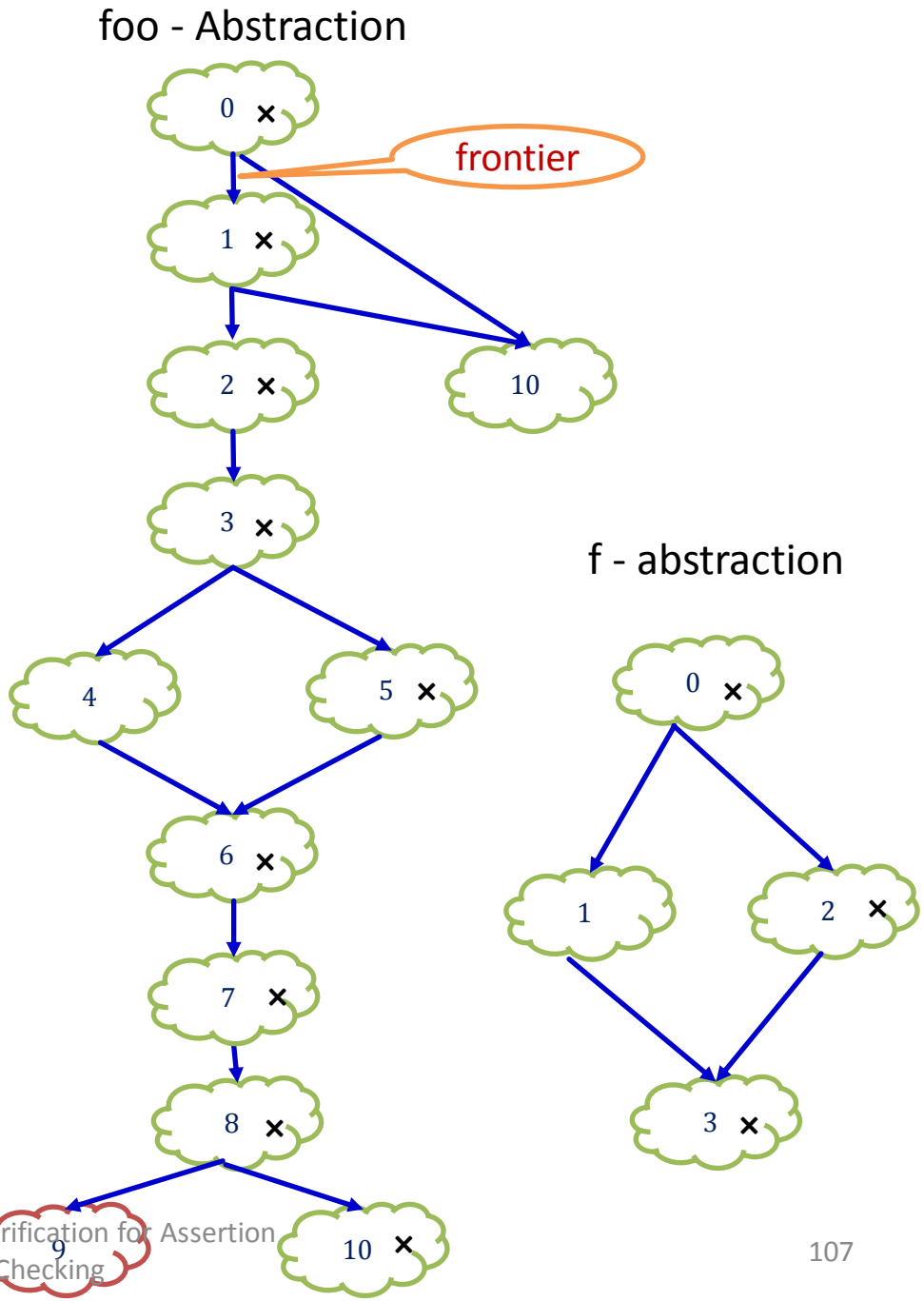
Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3, 4, 6, 7, 8, 9> and
Frontier is <0-1>

Step4: Suitable test is
<x, y, z, a > = <Random,…, 20>

Step5: Concrete trace is
<0, 1, 2, 3, 5, 6, 7, 8, 10 >

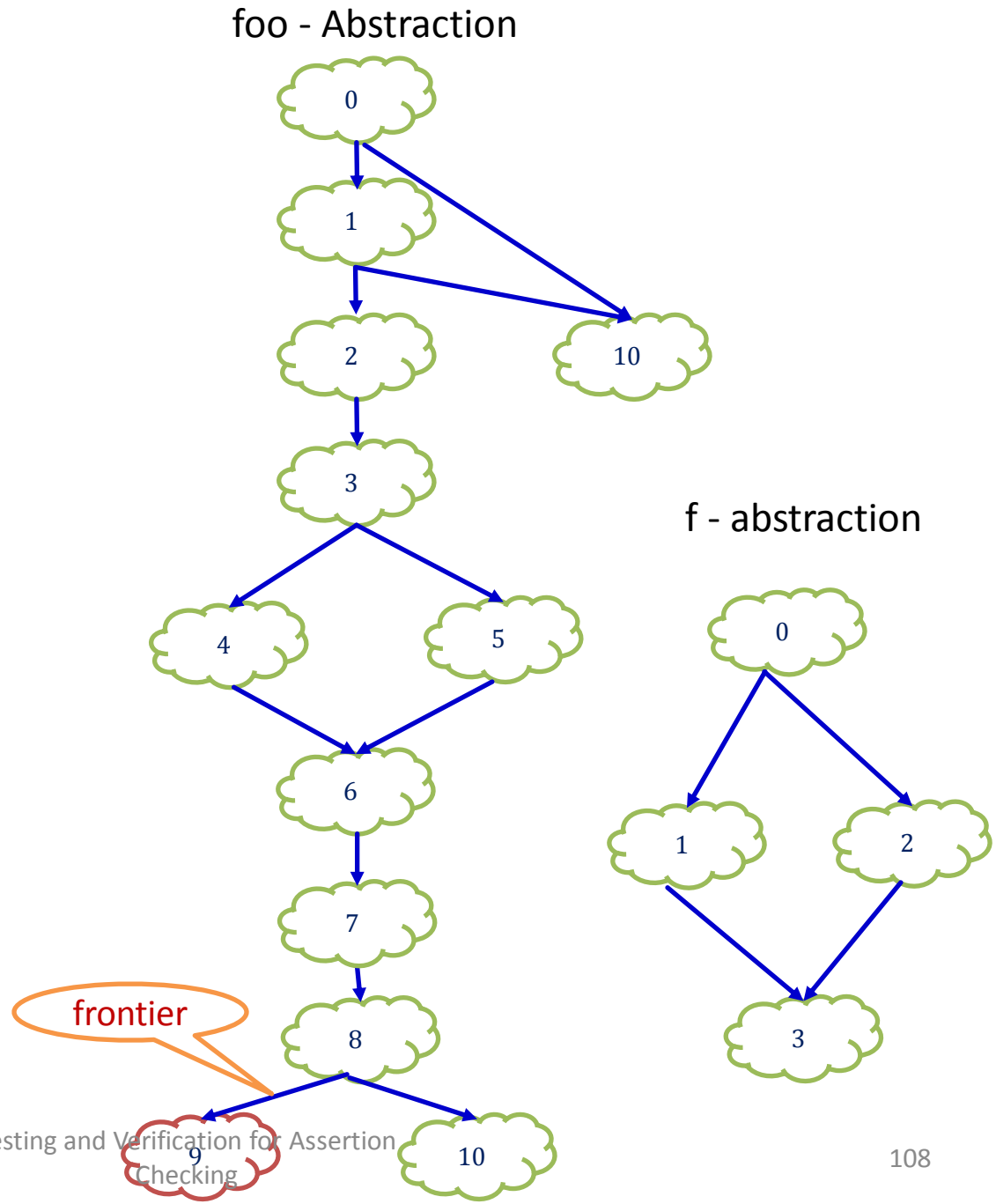


Iteration – 2

Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3, 5, 6, 7, 8, 9> and
Frontier is <8-9>



Iteration – 2

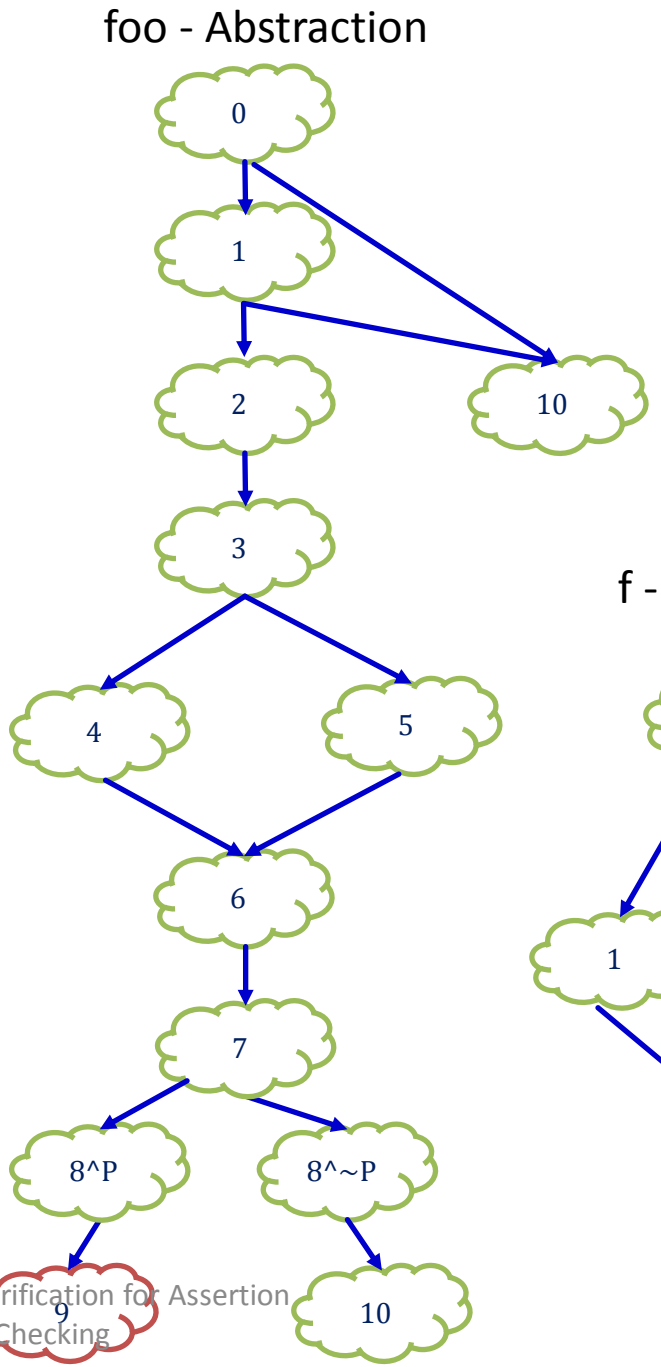
Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3, 5, 6, 7, 8, 9> and
Frontier is <8-9>

Step4: Couldn't generate a suitable test

Step5: Refining 8 into 8^P and 8^~P
Where P = (*x + *y + *z = 5)

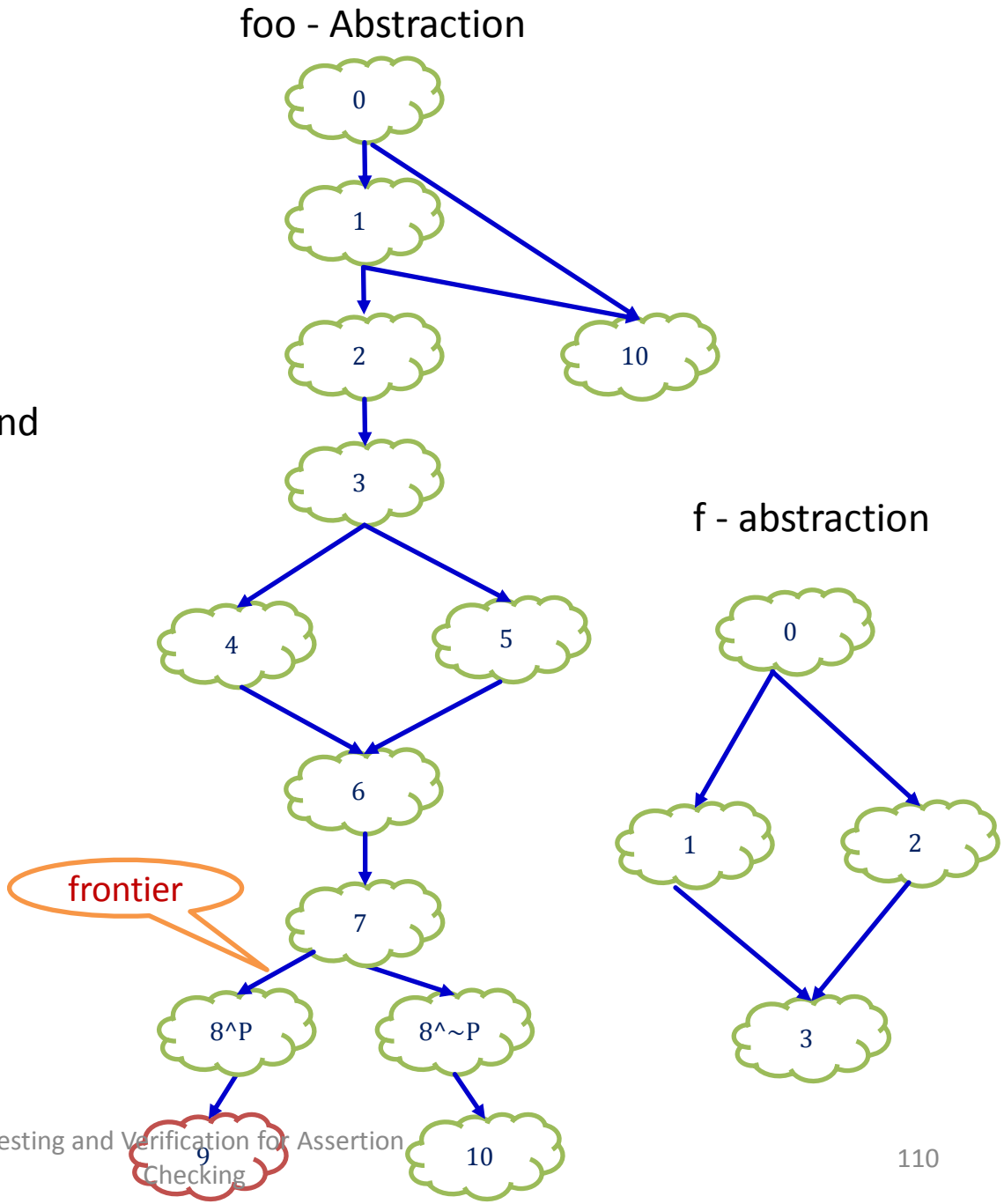


Iteration – 3

Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3, 5, 6, 7, 8^P, 9> and
Frontier is <7-(8^P)>



Iteration – 3

Step1: No witness

Step2: No proof

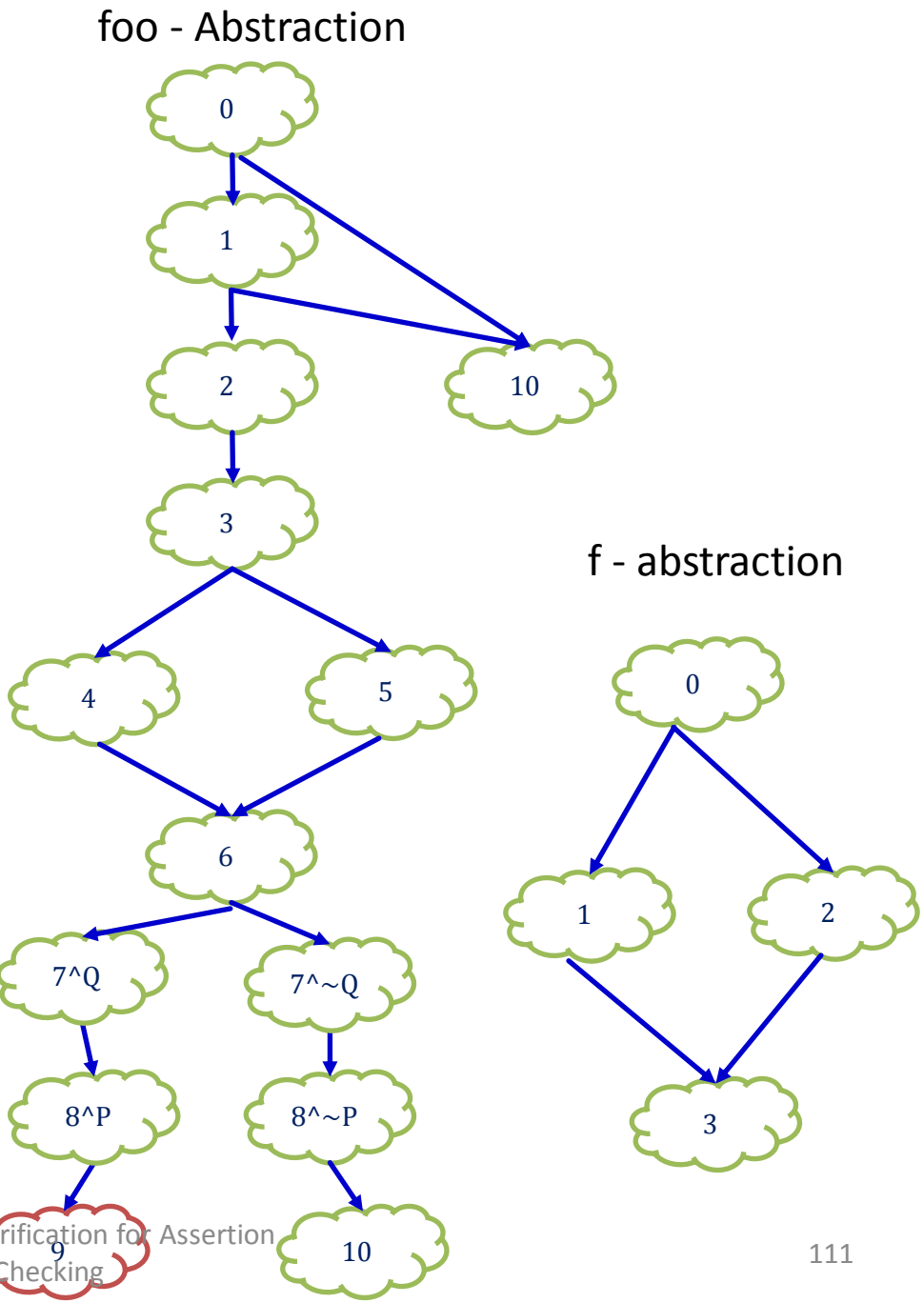
Step3: Abstract error trace is
<0, 1, 2, 3, 5, 6, 7, 8^P, 9> and
Frontier is <7-(8^P)>

Step4: Couldn't generate a suitable test

Step5: Refining 7 into 7^Q and 7^~Q

WP = (x = y & x != z) & (*x + *y = 3)

WP(alpha)
= ~ ((x = y & x != z) & ~WP)
= ~ (x = y & x != z) | (*x + *y = 3)
= Q

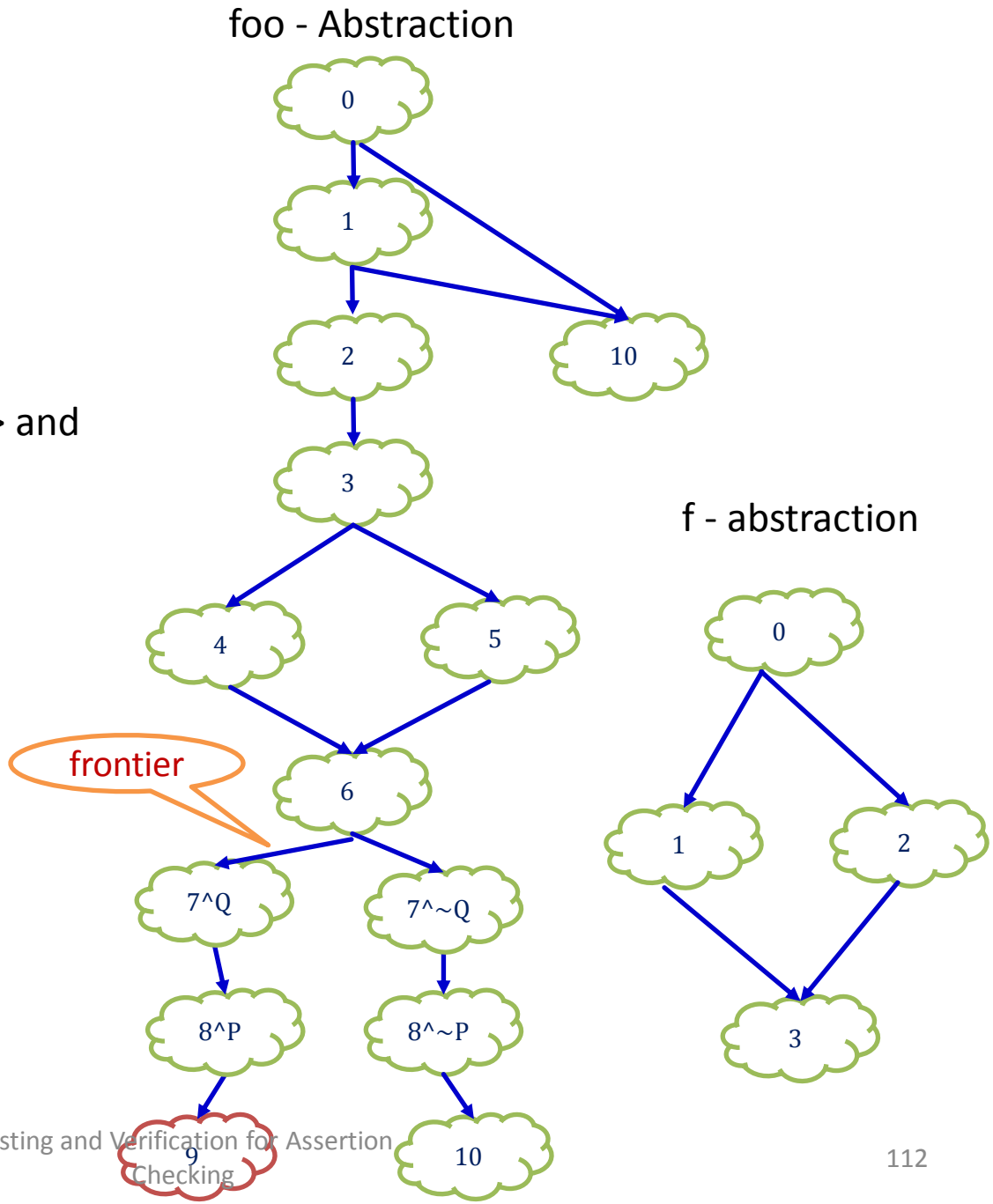


Iteration – 4

Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3, 5, 6, 7^Q, 8^P, 9> and
Frontier is <6-(7^Q)>



Iteration – 4

Step1: No witness

Step2: No proof

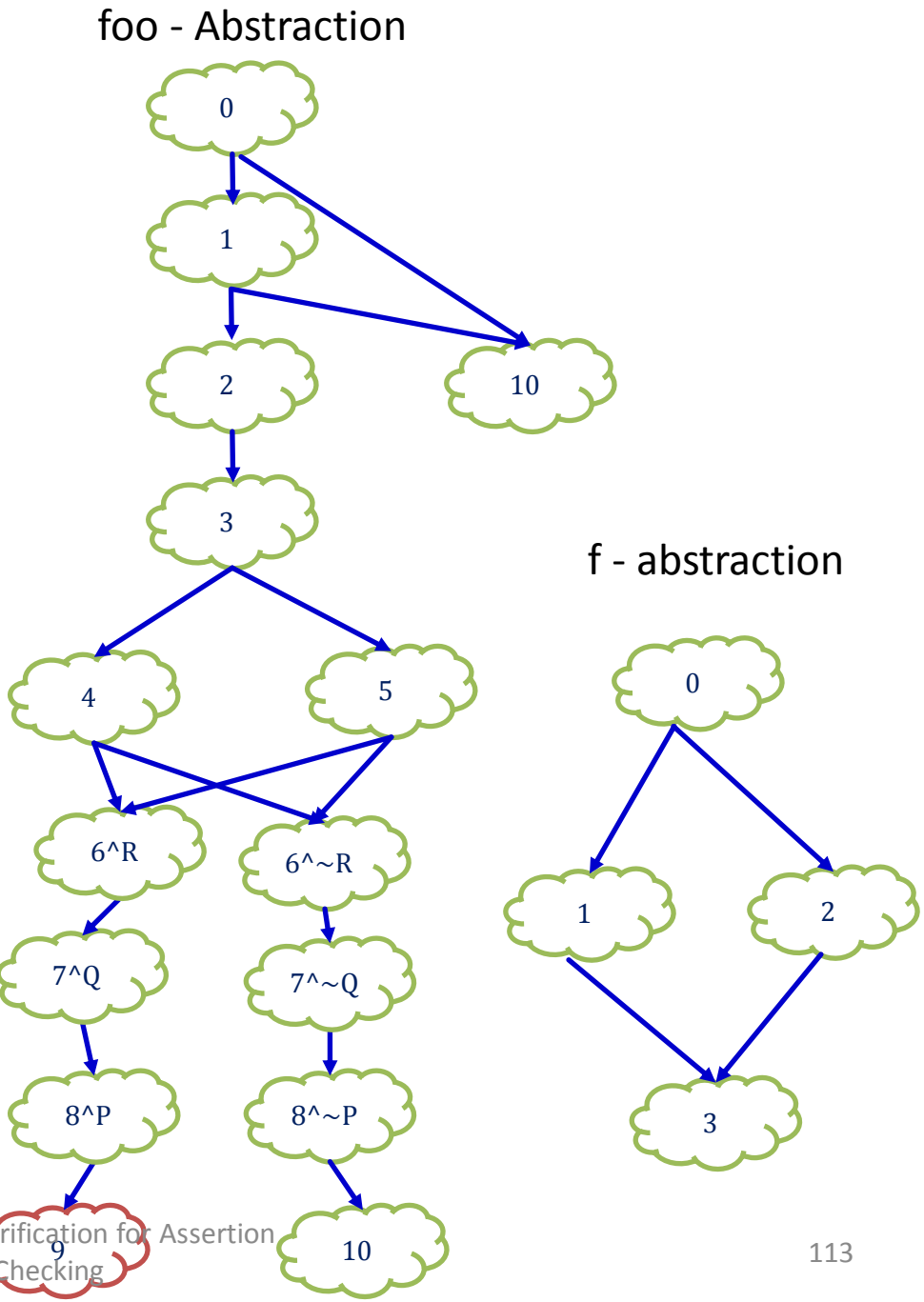
Step3: Abstract error trace is
<0, 1, 2, 3, 5, 6, 7^Q, 8^P, 9> and
Frontier is <6-(7^Q)>

Step4: Couldn't generate a test input

Step5: Refining 6 into 6^R and 6^~R

WP = (x = y & x != z) & (*x = 2)

WP(alpha)
= ~ ((x = y & x != z) & ~WP)
= ~ (x = y & x != z) | (*x = 2)
= R

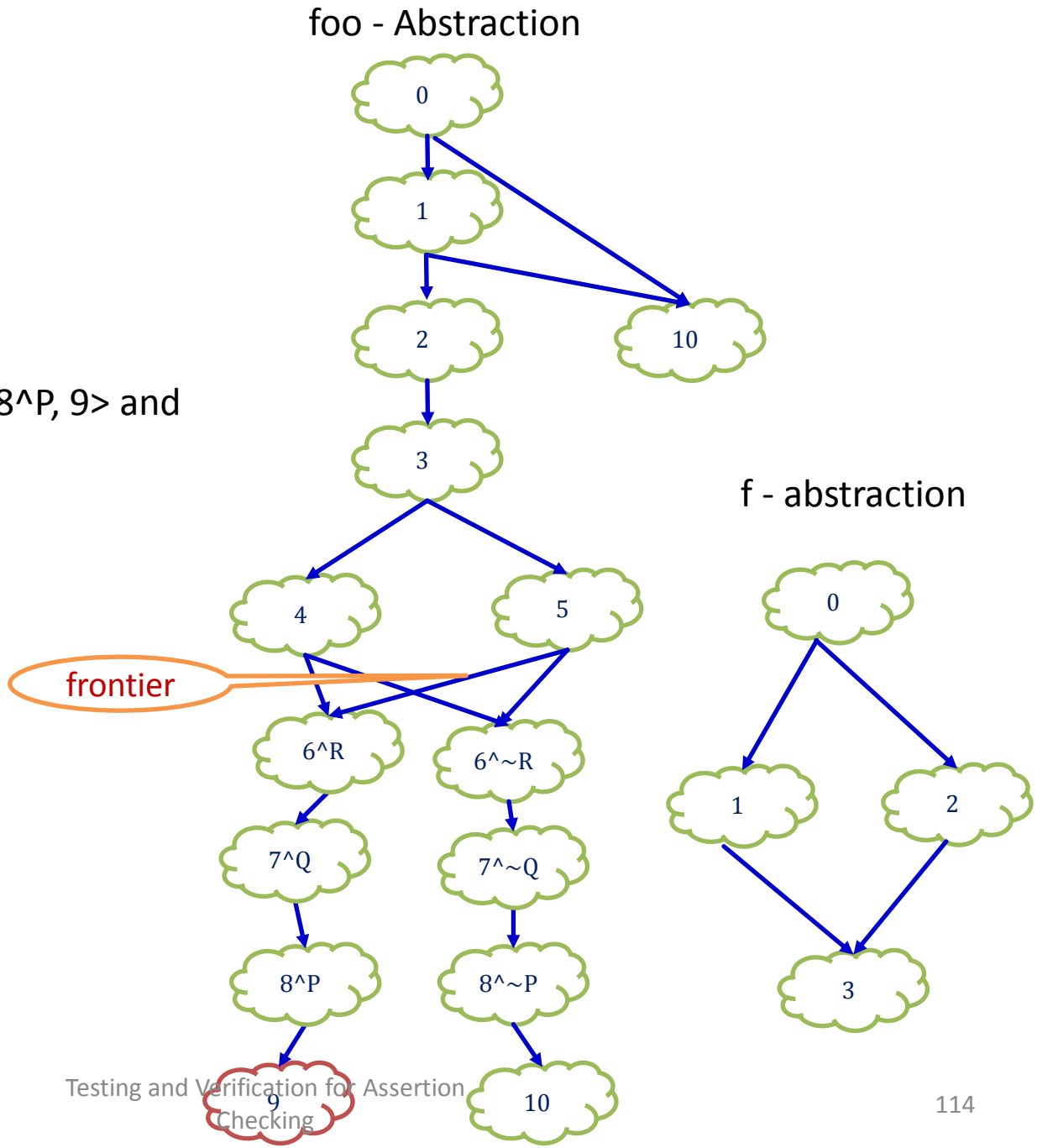


Iteration – 5

Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3, 5, 6^R, 7^Q, 8^P, 9> and
Frontier is <5-(6^R)>



Iteration – 5

Step1: No witness

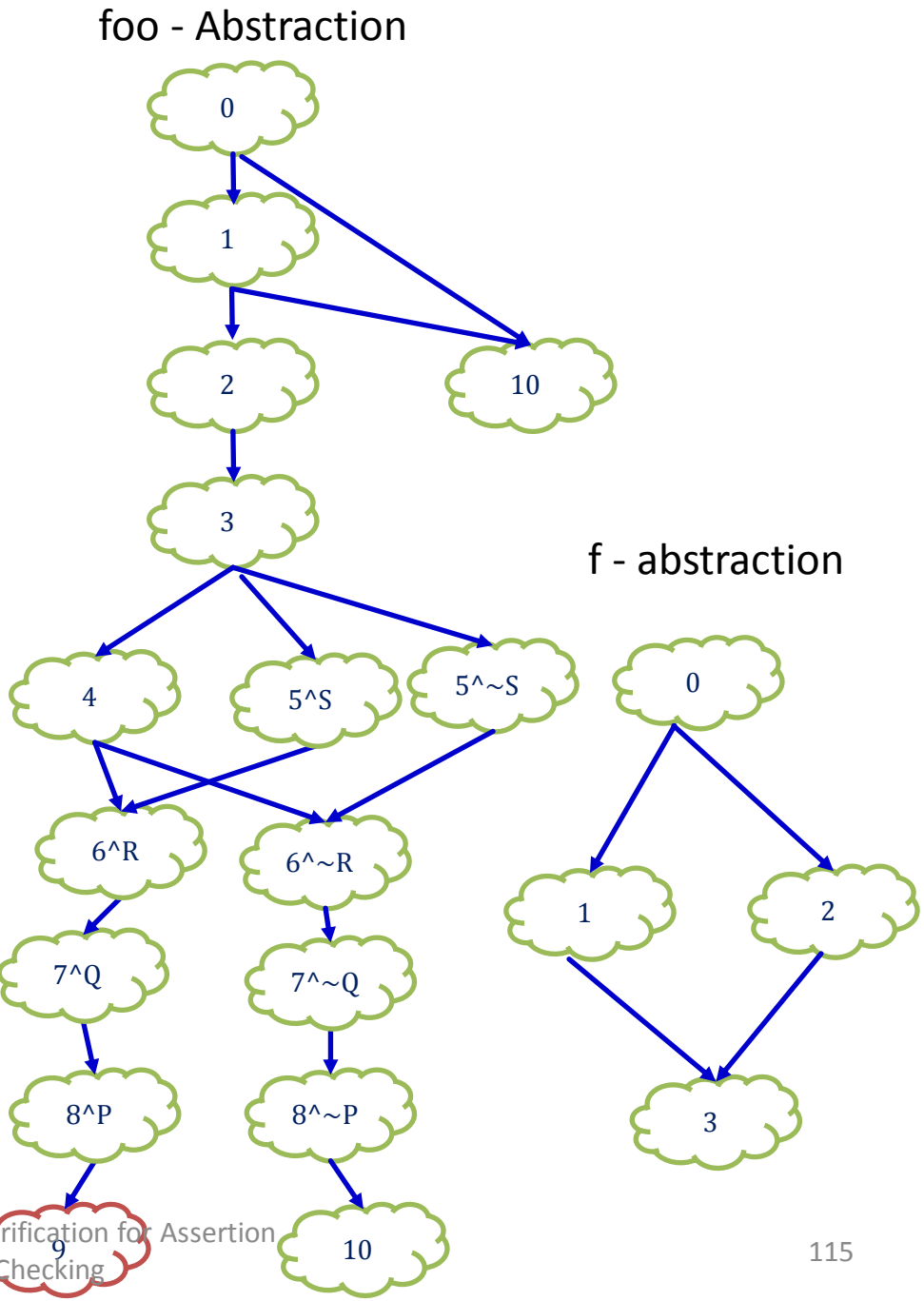
Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3, 5, 6^R, 7^Q, 8^P, 9> and
Frontier is <5-(6^R)>

Step4: Couldn't generate test input

Step5: Refining 5 into 5^S and 5^~S

Where s = ~(x = z) | (*x = 2)

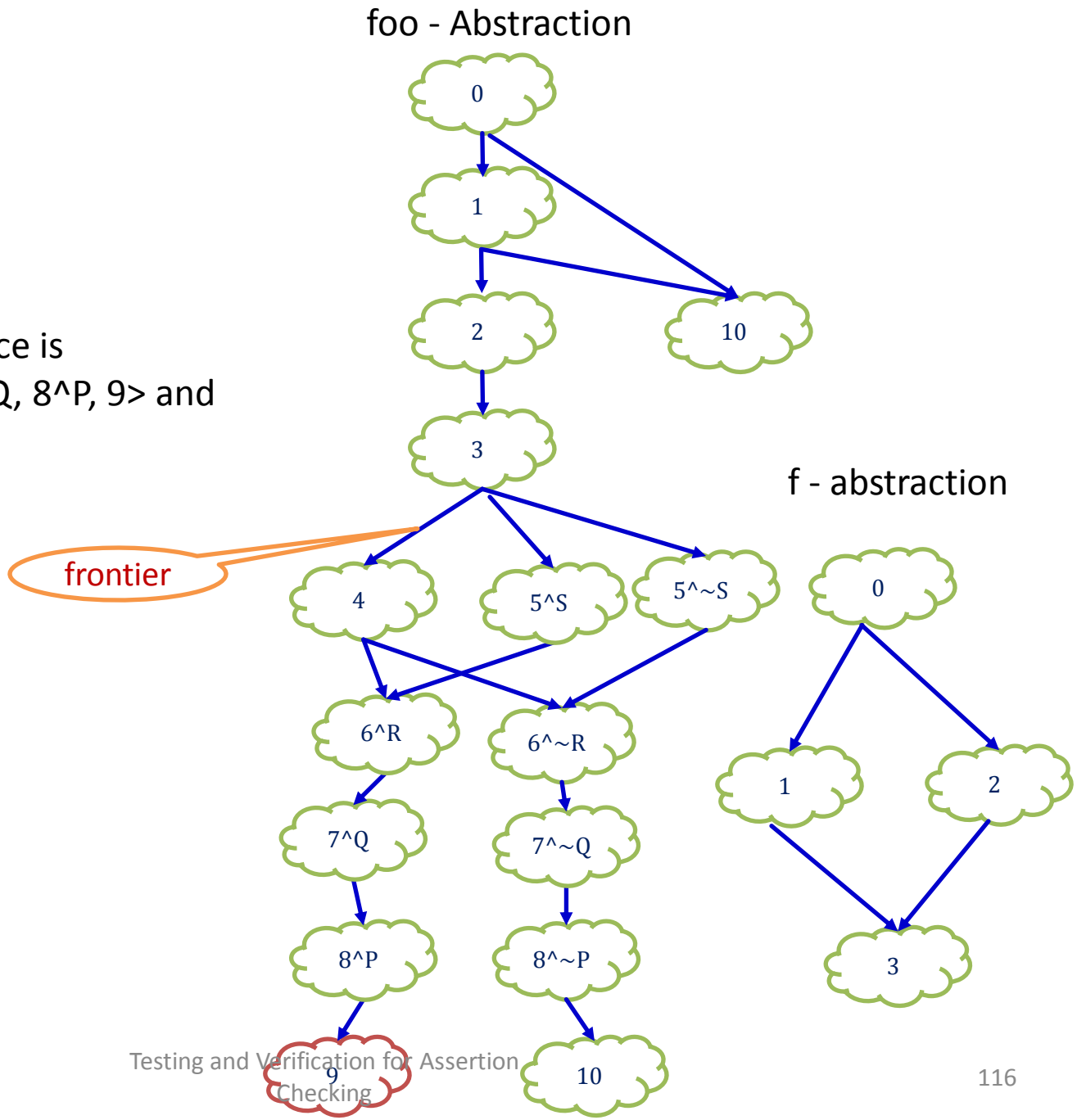


Iteration – 6

Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3, 4, 6^R, 7^Q, 8^P, 9> and
Frontier is <3-4>



Iteration – 6

Step1: No witness

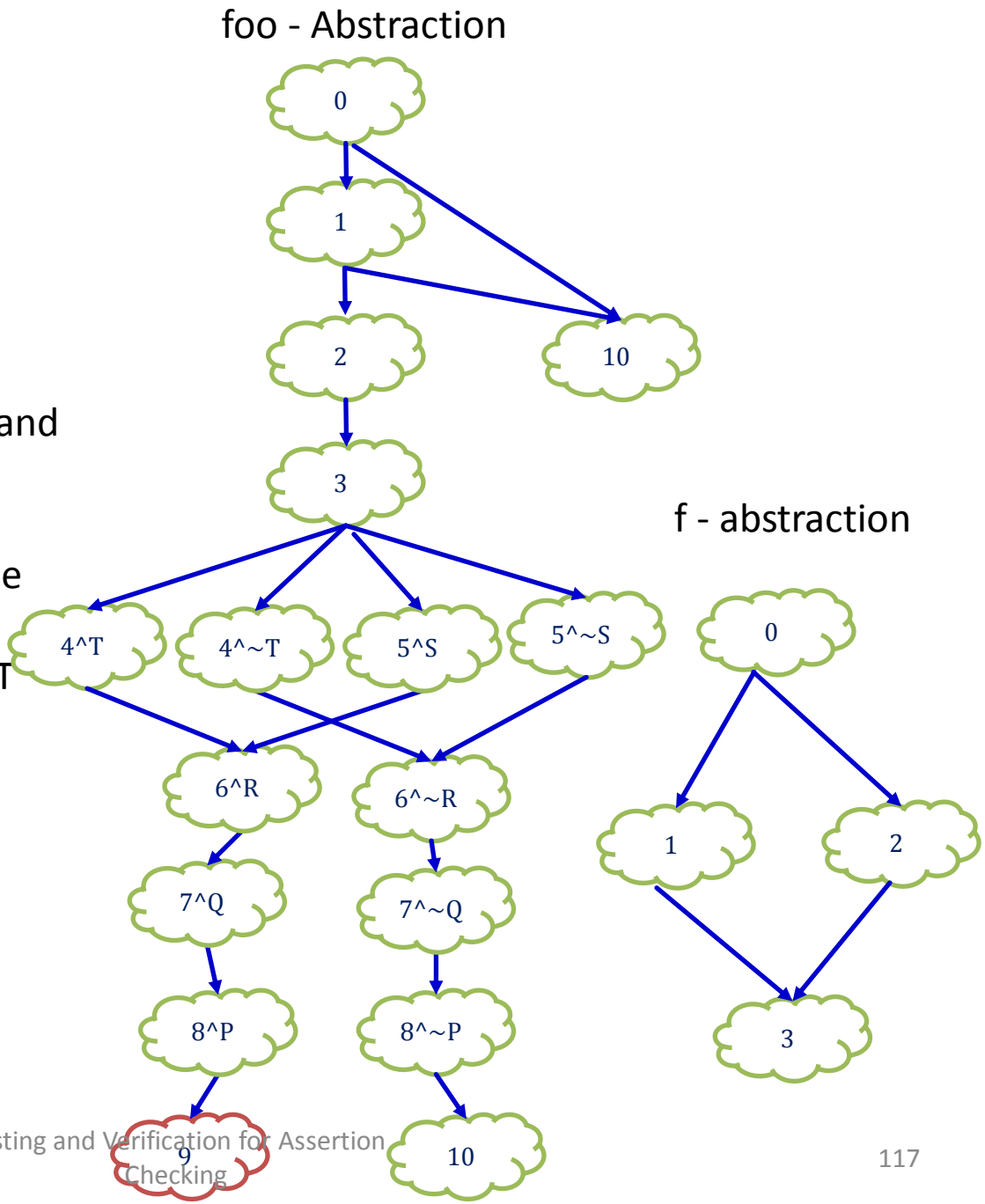
Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3, 4, 6^R, 7^Q, 8^P, 9> and
Frontier is <3-4>

Step4: Couldn't generate a test case

Step5: Splitting 4 into 4^T and 4^~T

Where T = ~(x = y) | (*x = 2)

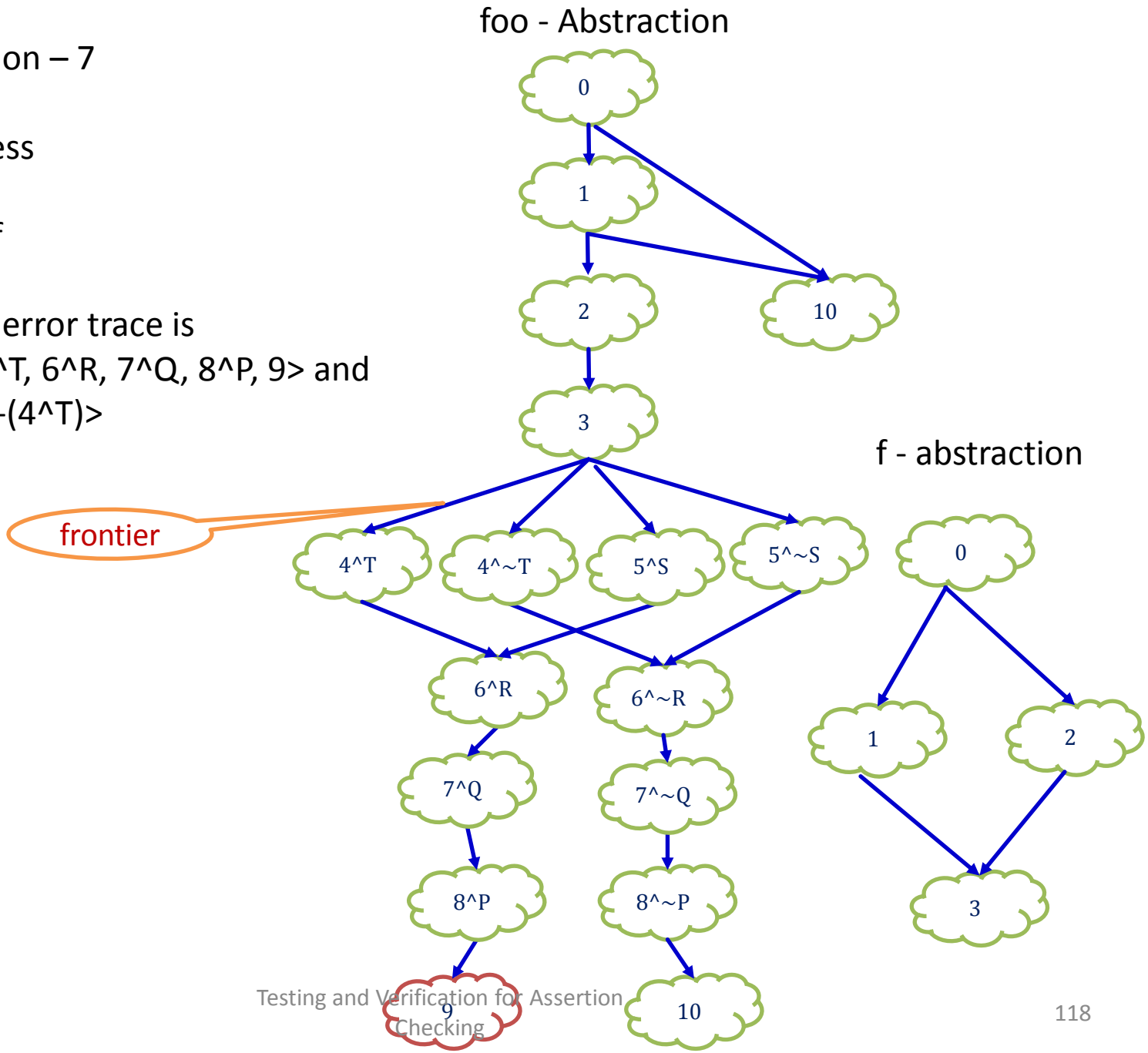


Iteration – 7

Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3, 4^T, 6^R, 7^Q, 8^P, 9> and
Frontier is <3-(4^T)>



Iteration – 7

Step1: No witness

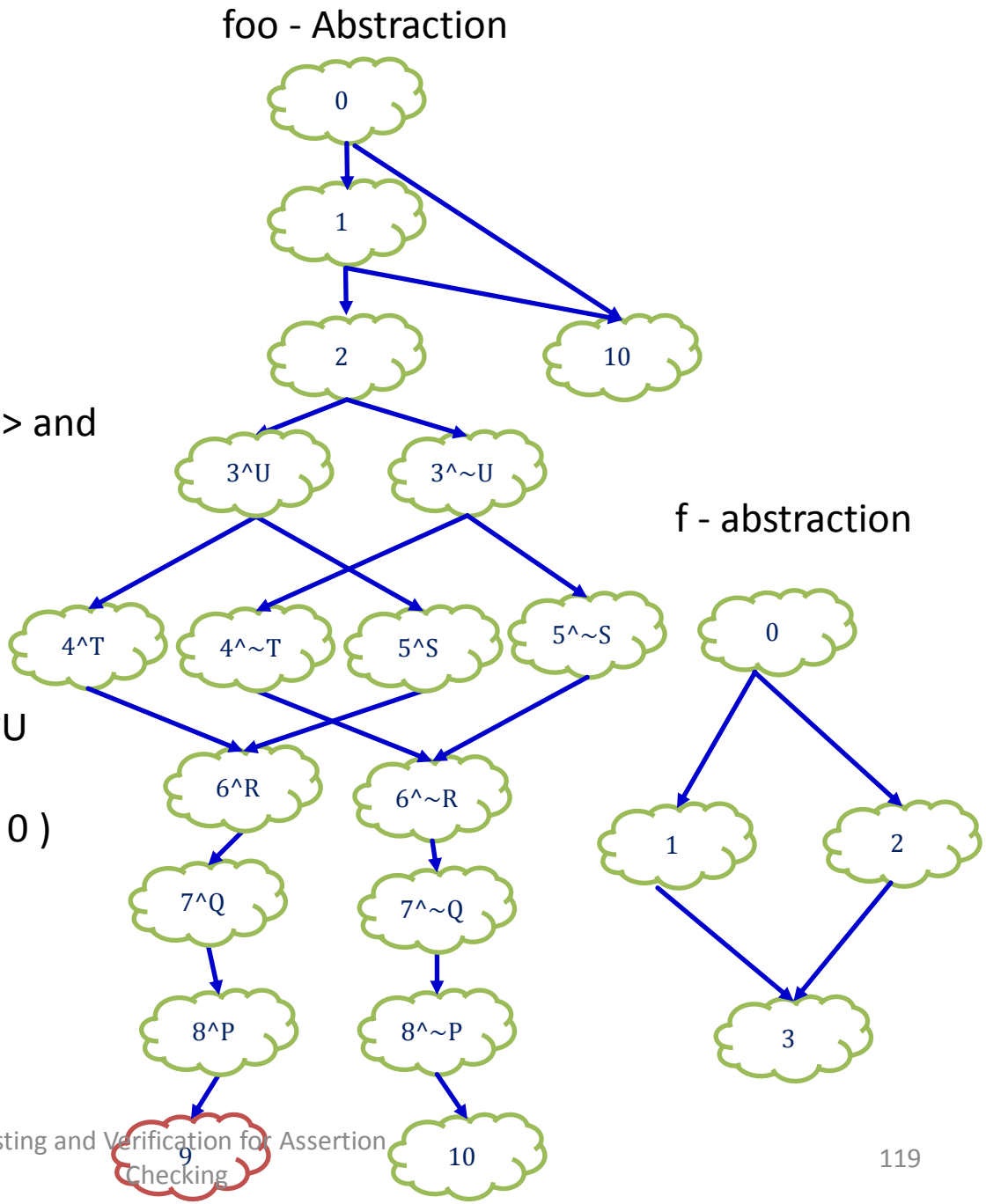
Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3, 4^T, 6^R, 7^Q, 8^P, 9> and
Frontier is <3-(4^T)>

Step4: Couldn't generate test

Step5: Splitting 3 into 3^U and 3^~U

Where U = ~(x!= y & x!= z) | (ret < 0)

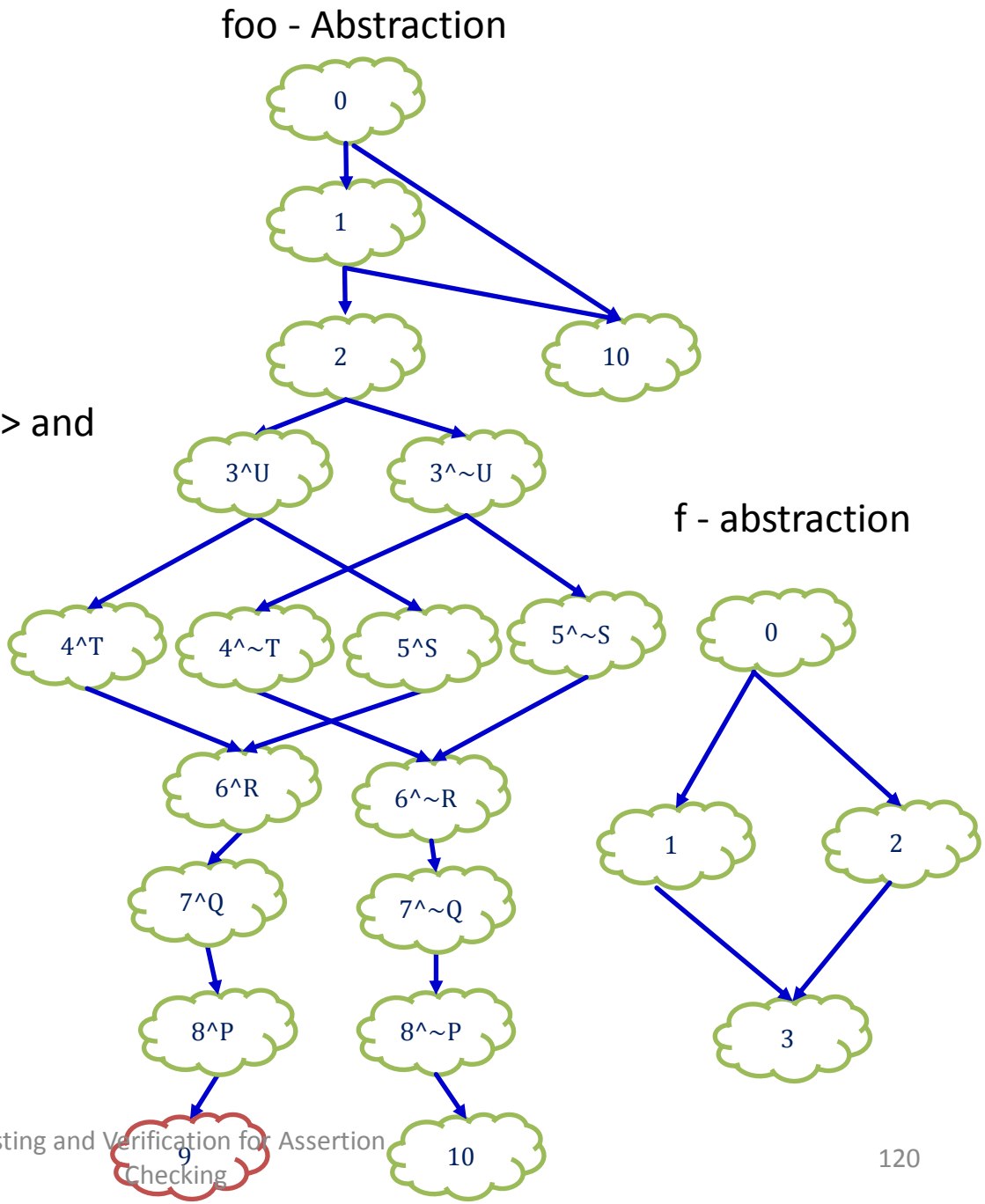


Iteration – 8

Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3^U, 4^T, 6^R, 7^Q, 8^P, 9> and
Frontier is <2-(3^U)>



Iteration – 8

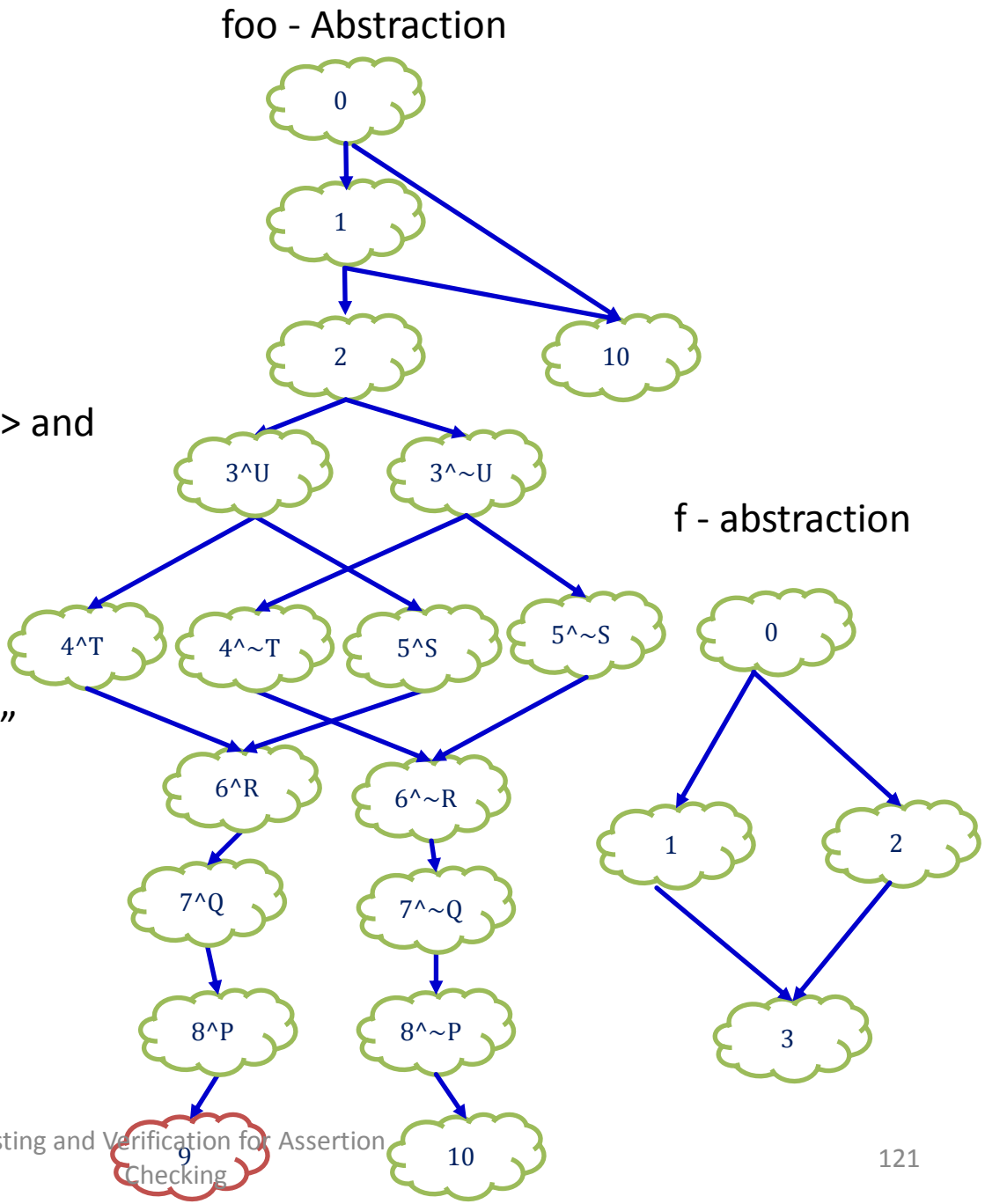
Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3^U, 4^T, 6^R, 7^Q, 8^P, 9> and
Frontier is <2-(3^U)>

Step4: Inter procedure call

Calling DASH recursively with conditions on initial states as “true” and on final states as “ a < 0” (Since ret < 0)



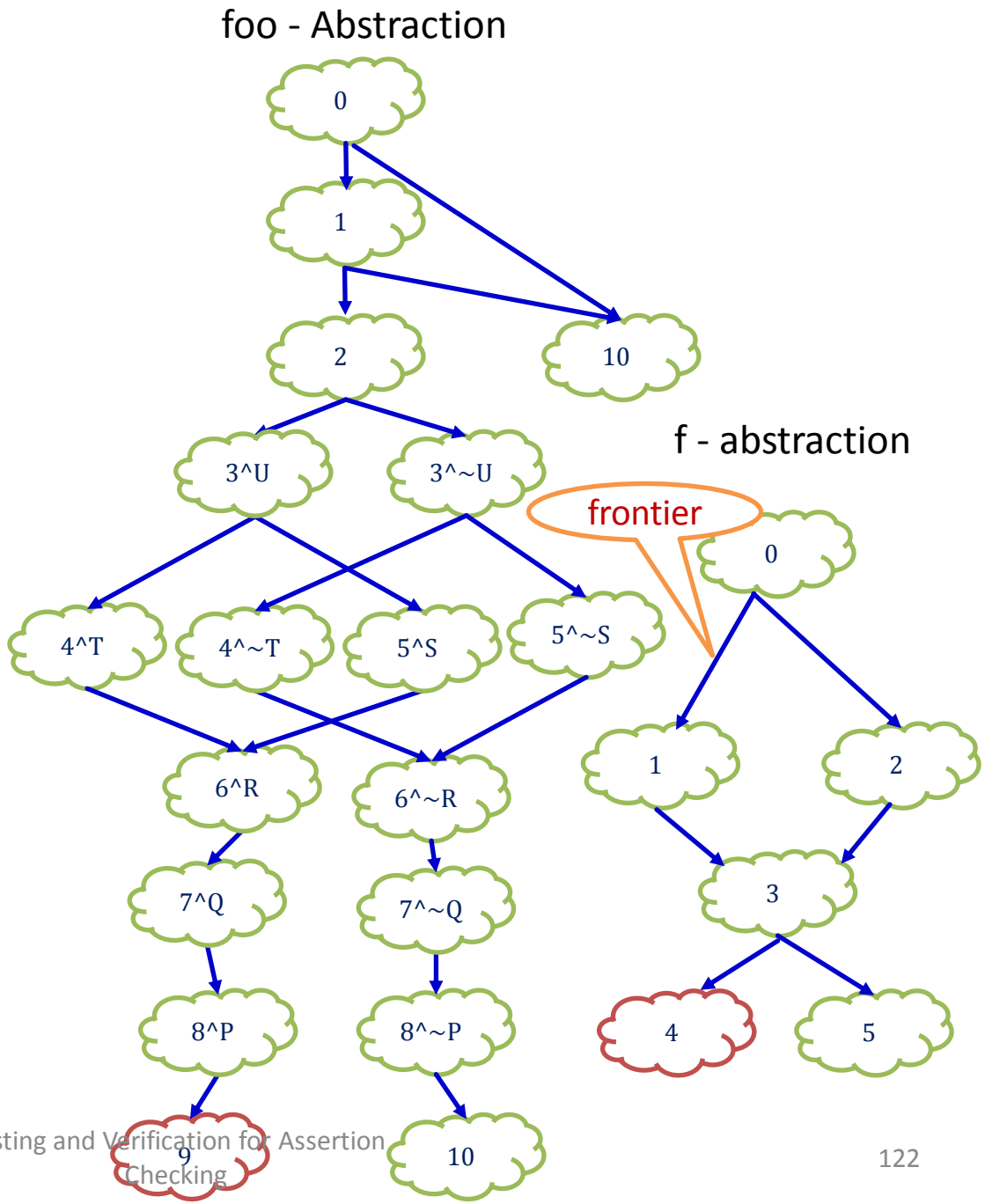
Iteration – 8 - 1

Recursive call – 1

Step1: No witness

Step2: No proof

Step3: Abstract error trace is <0, 1, 3, 4> and Frontier is <0-1>



Iteration – 8 - 1

Recursive call – 1

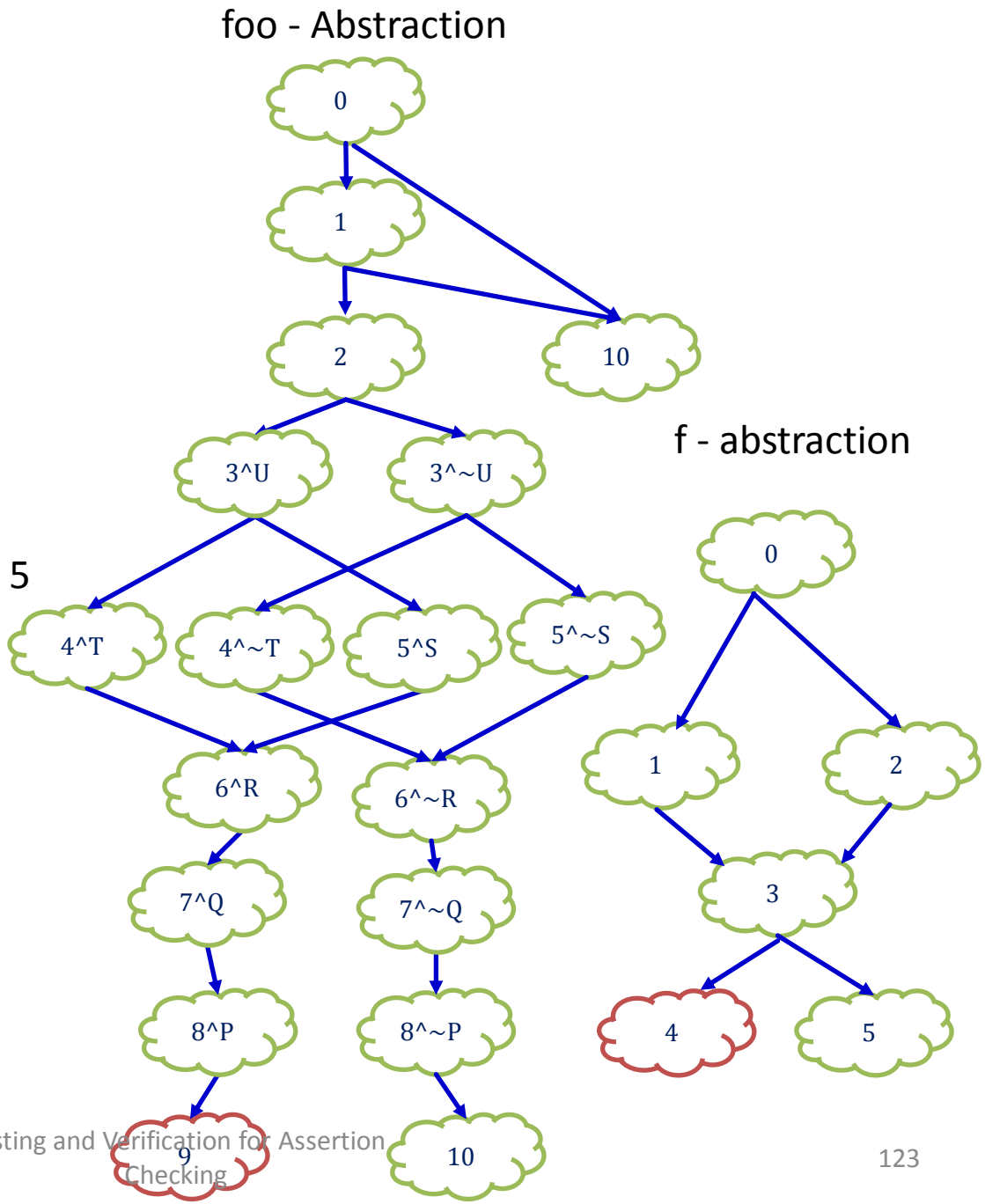
Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 3, 4> and Frontier is <0-1>

Step4: Could generate an input a = 5

Step5: Concrete trace is
<0, 2, 3, 5 >



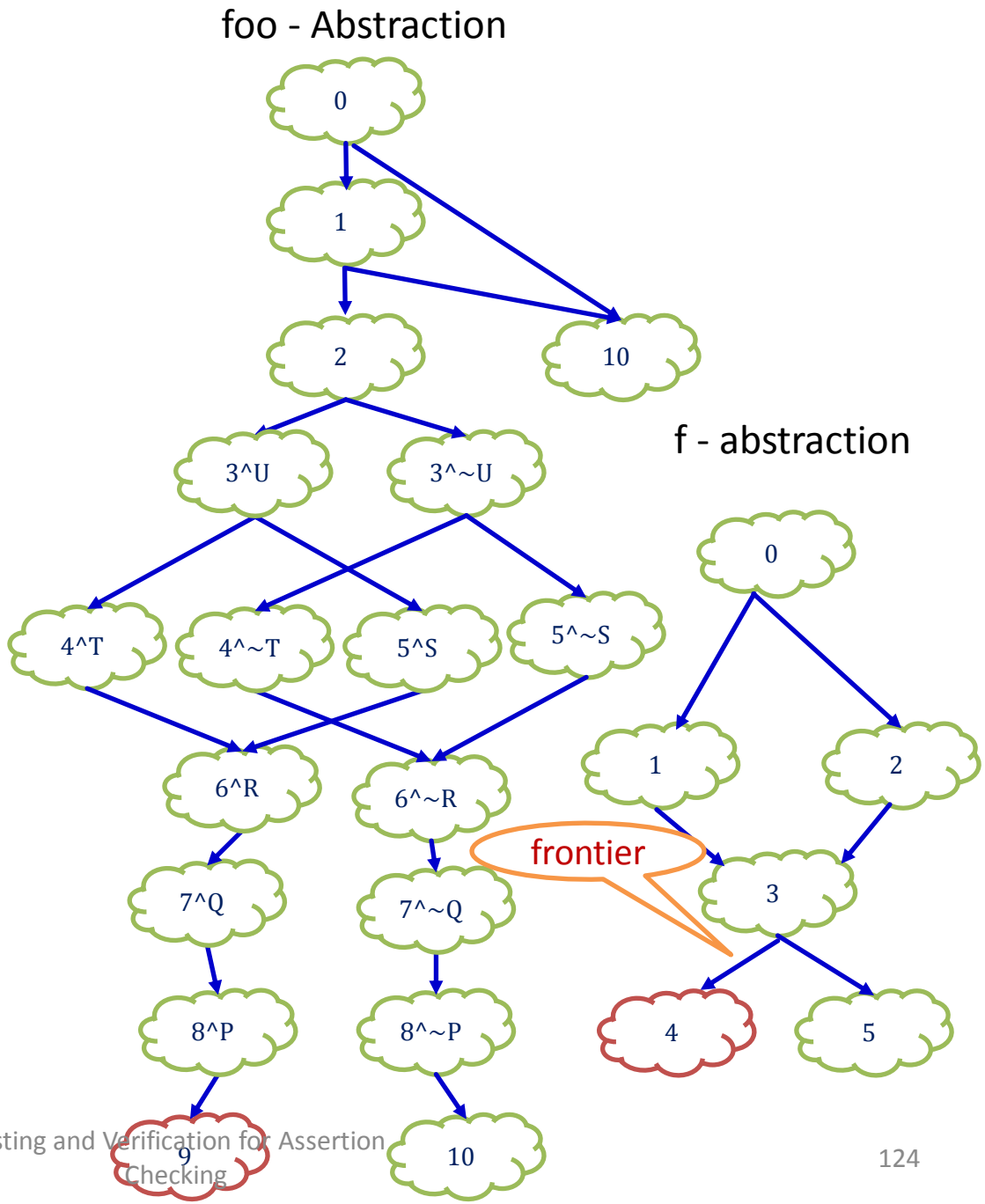
Iteration – 8 - 2

Recursive call – 1

Step1: No witness

Step2: No proof

Step3: Abstract error trace is <0, 2, 3, 4> and Frontier is <3-4>



Iteration – 8 - 2

Recursive call – 1

Step1: No witness

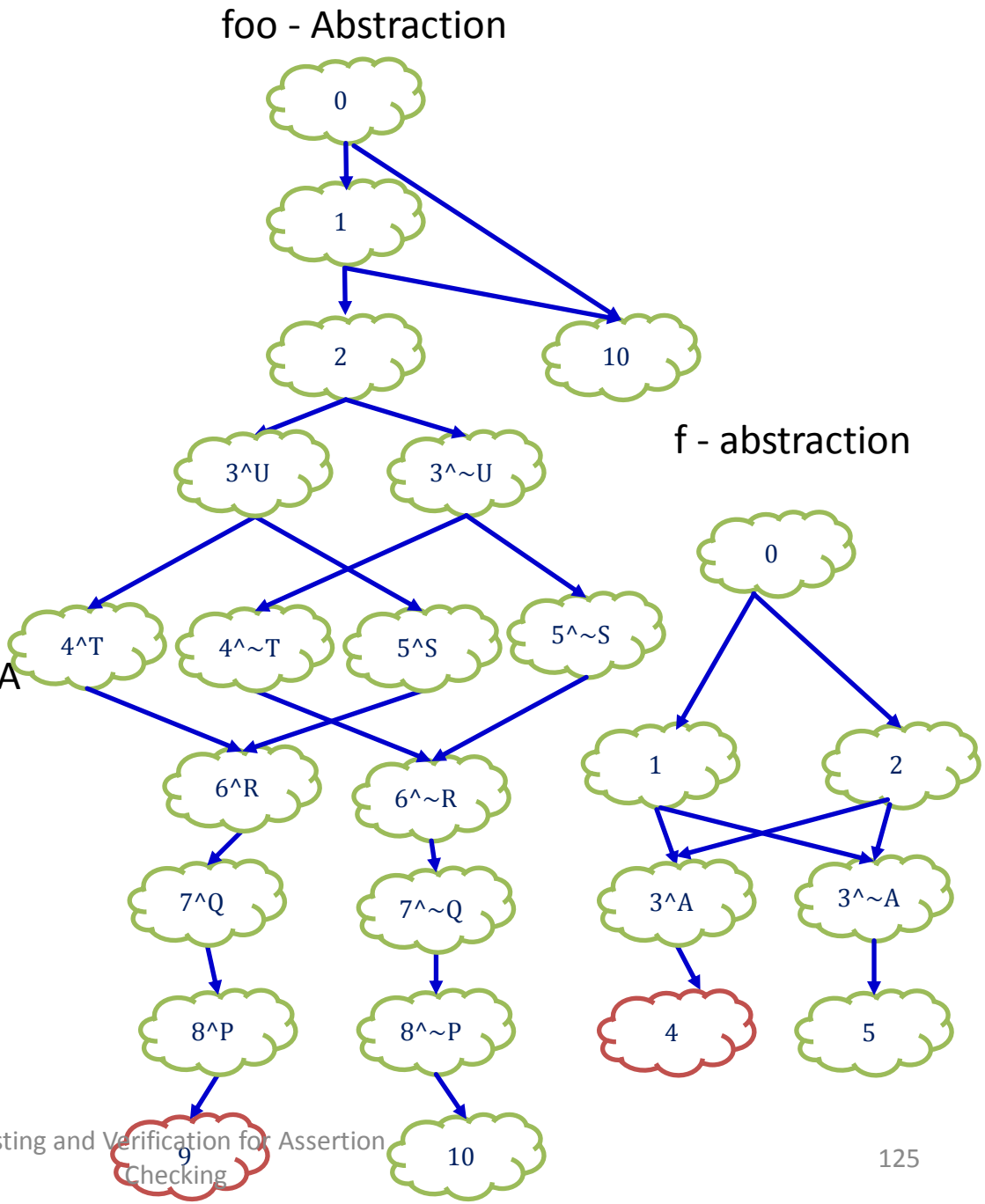
Step2: No proof

Step3: Abstract error trace is <0, 2, 3, 4> and Frontier is <3-4>

Step4: Couldn't generate input

Step5: Refining 3 into 3^A and 3^~A

Where A = (a < 0)



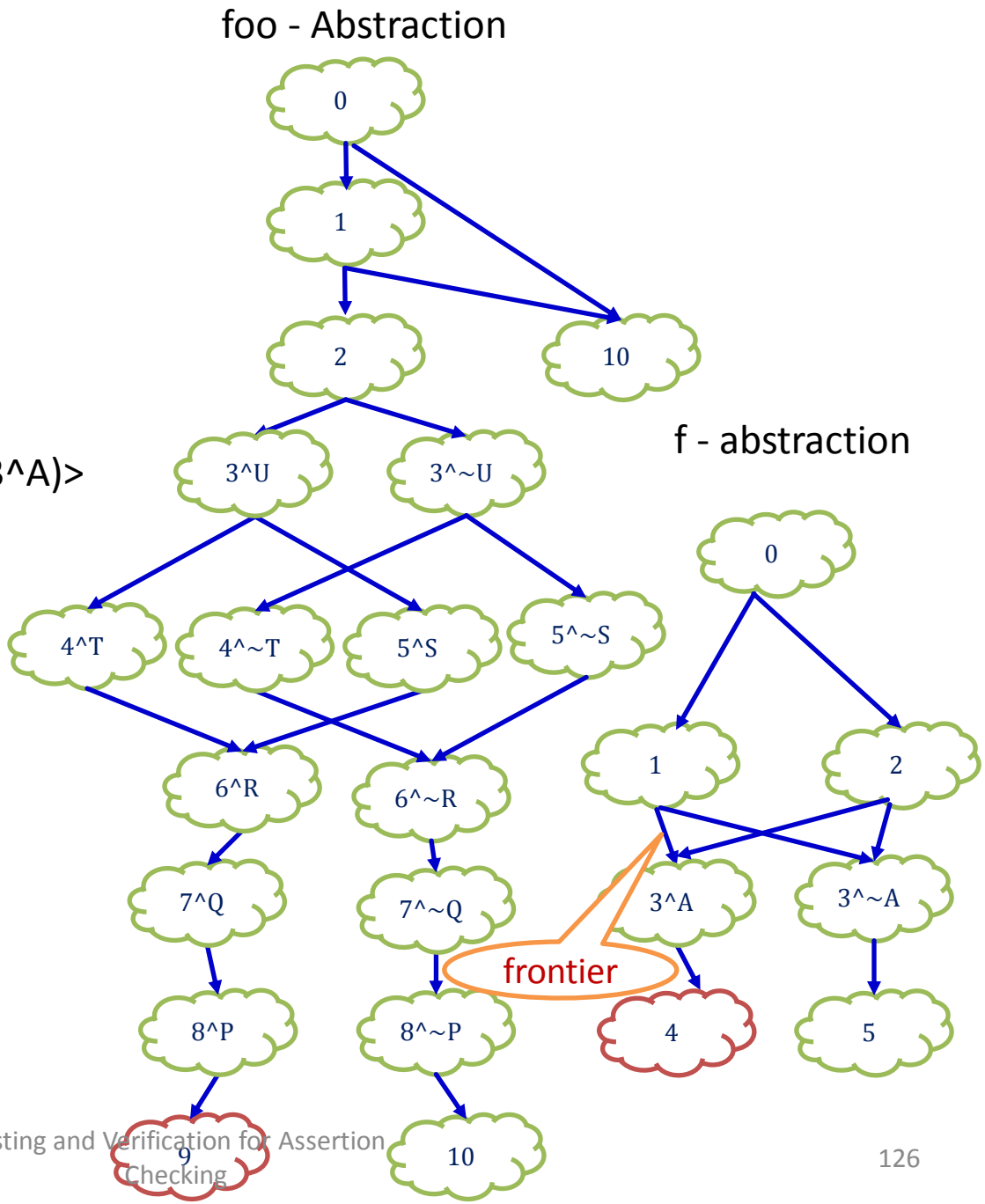
Iteration – 8 - 3

Recursive call – 1

Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 1, 3^A, 4> and Frontier is <1-(3^A)>



Iteration – 8 - 3

Recursive call – 1

Step1: No witness

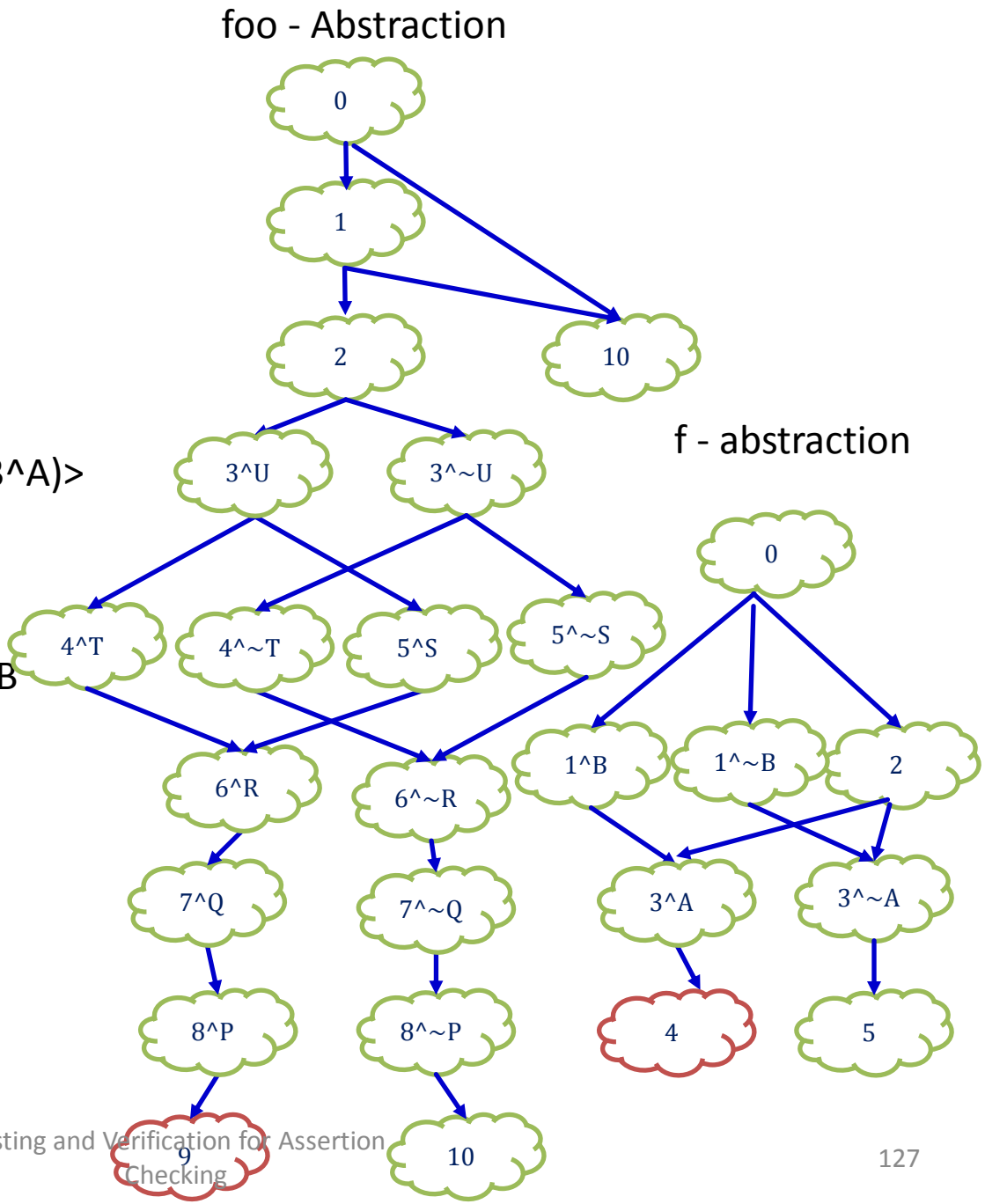
Step2: No proof

Step3: Abstract error trace is
<0, 1, 3^A, 4> and Frontier is <1-(3^A)>

Step4: Couldn't generate an input

Step5: Refining 1 into 1^B and 1^~B

Where B = a < 0



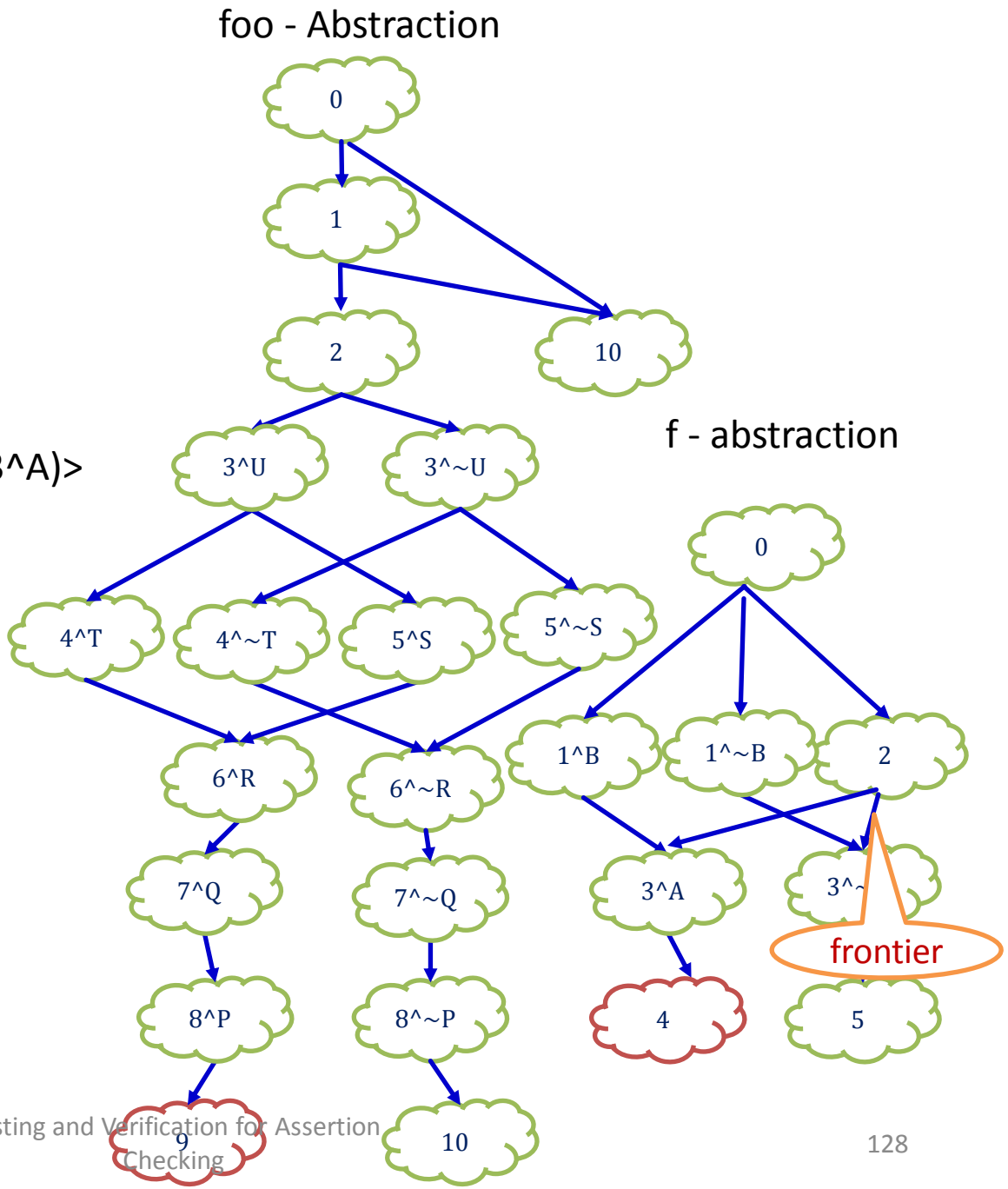
Iteration – 8 - 4

Recursive call – 1

Step1: No witness

Step2: No proof

Step3: Abstract error trace is
<0, 2, 3^A, 4> and Frontier is <2-(3^A)>



Iteration – 8 - 4

Recursive call – 1

Step1: No witness

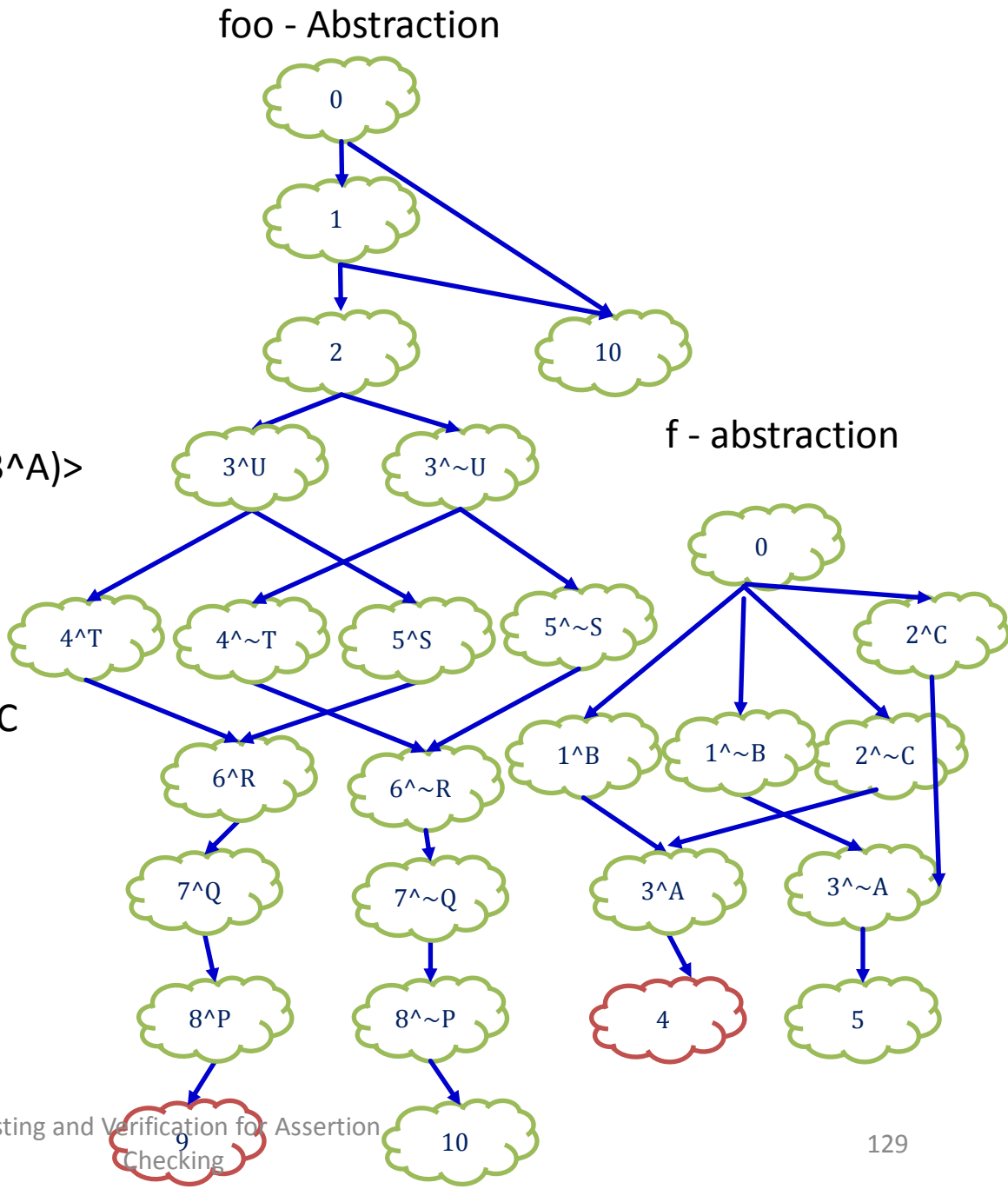
Step2: No proof

Step3: Abstract error trace is
<0, 2, 3^A, 4> and Frontier is <2-(3^A)>

Step4: Couldn't generate an input

Step5: Splitting 2 into 2^C and 2^~C

Where C = a > 0



Iteration – 8 - 5

Recursive call – 1

Step1: No witness

Step2: No proof

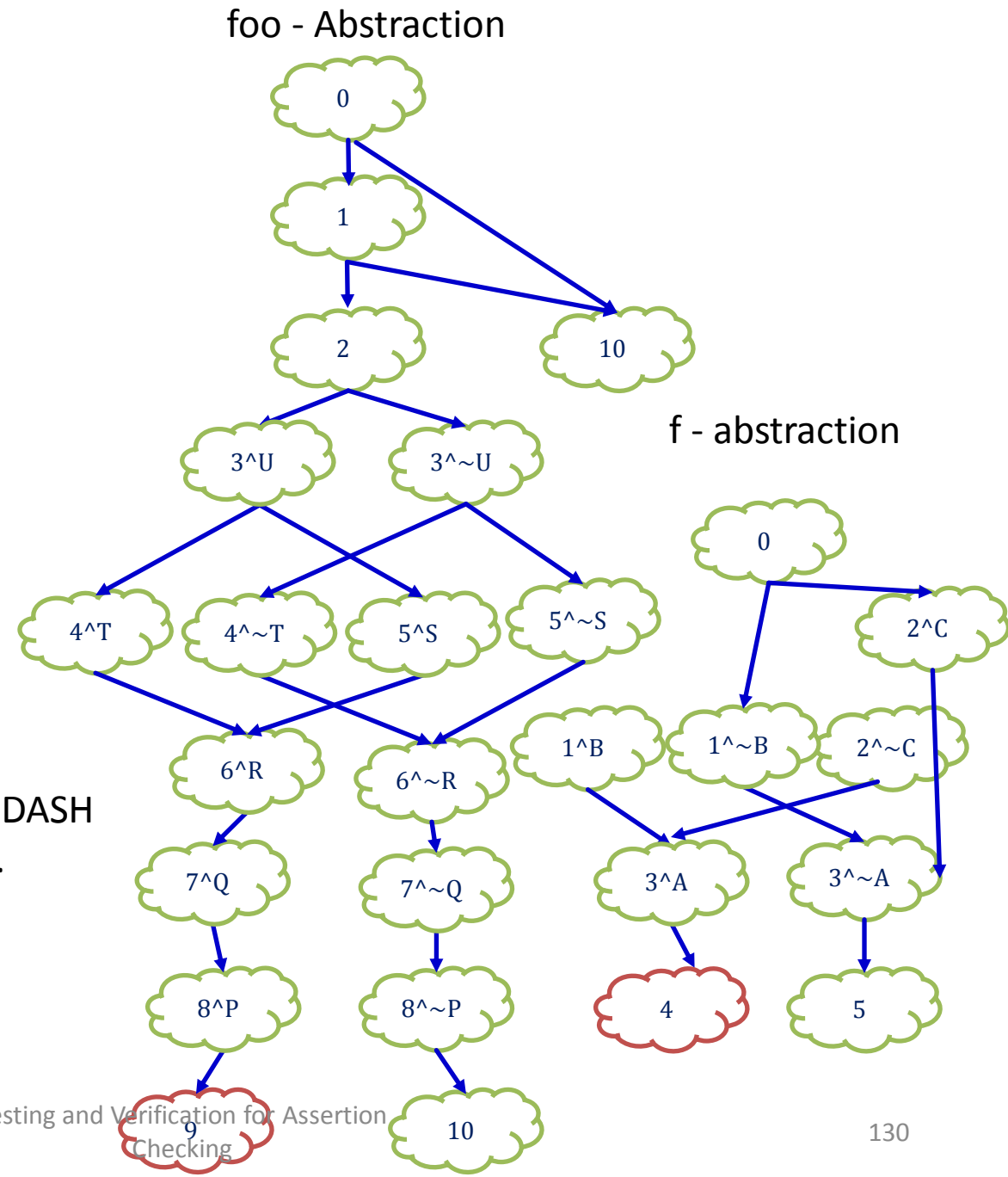
Step3: Abstract error trace is
<0, 2^C, 3^A, 4> and
Frontier is <0-(2^C)>

Step4: No input

Step5: Splitting

Finally, On recursive invocation of DASH returns proof for the f abstraction.

Compute Refine prediction
= ~ (a < 0 | a > 0) = false



Iteration – 8

Step1: No witness

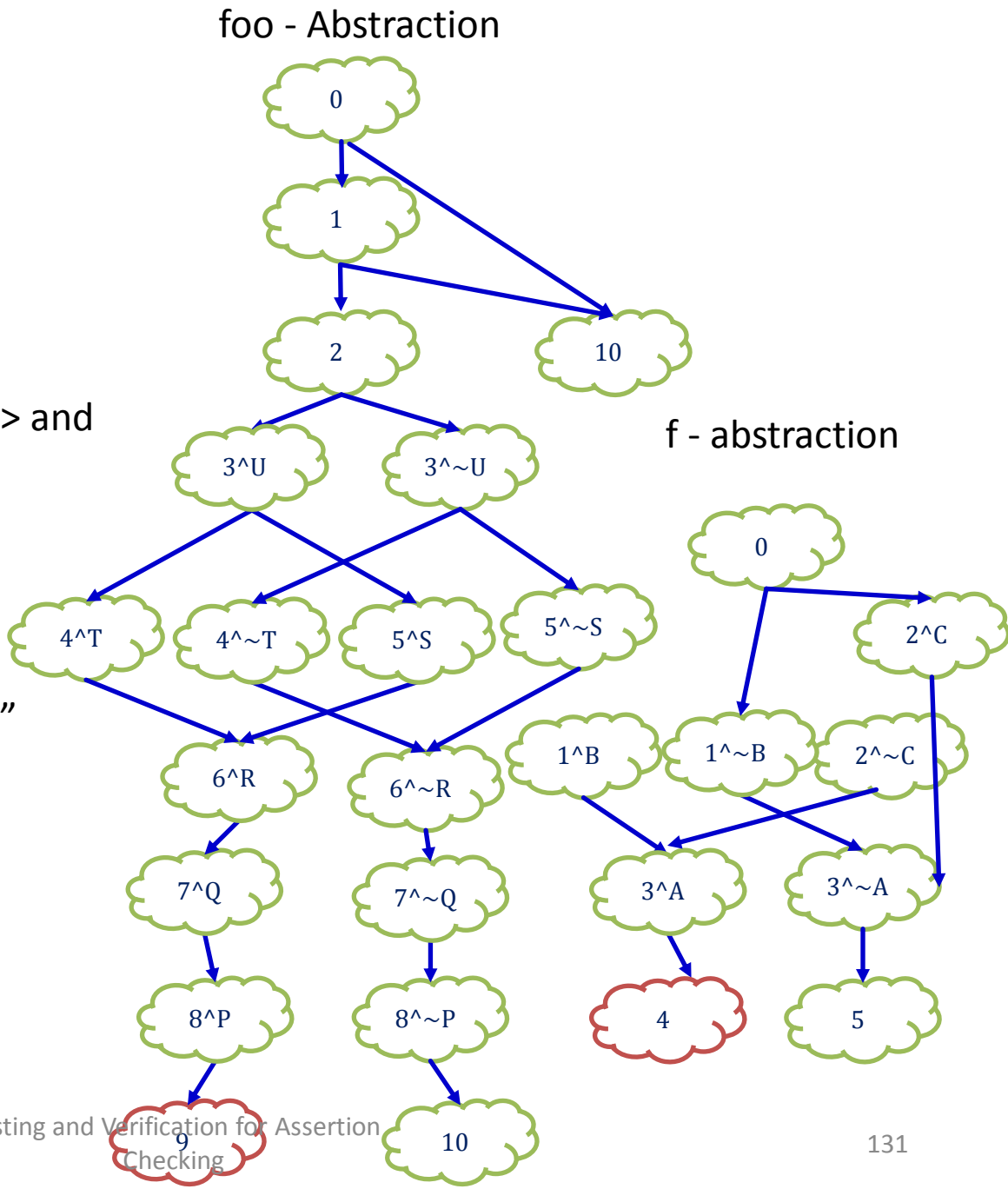
Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3^U, 4^T, 6^R, 7^Q, 8^P, 9> and
Frontier is <2-(3^U)>

Step4: Inter procedure call

Calling DASH recursively with conditions on initial states as “true” and on final states as “ a < 0” (Since ret < 0)
Compute Refine prediction
= ~ (a < 0 | a > 0) = false = V

Splitting 2 into 2^V and 2^~V



Iteration – 8

Step1: No witness

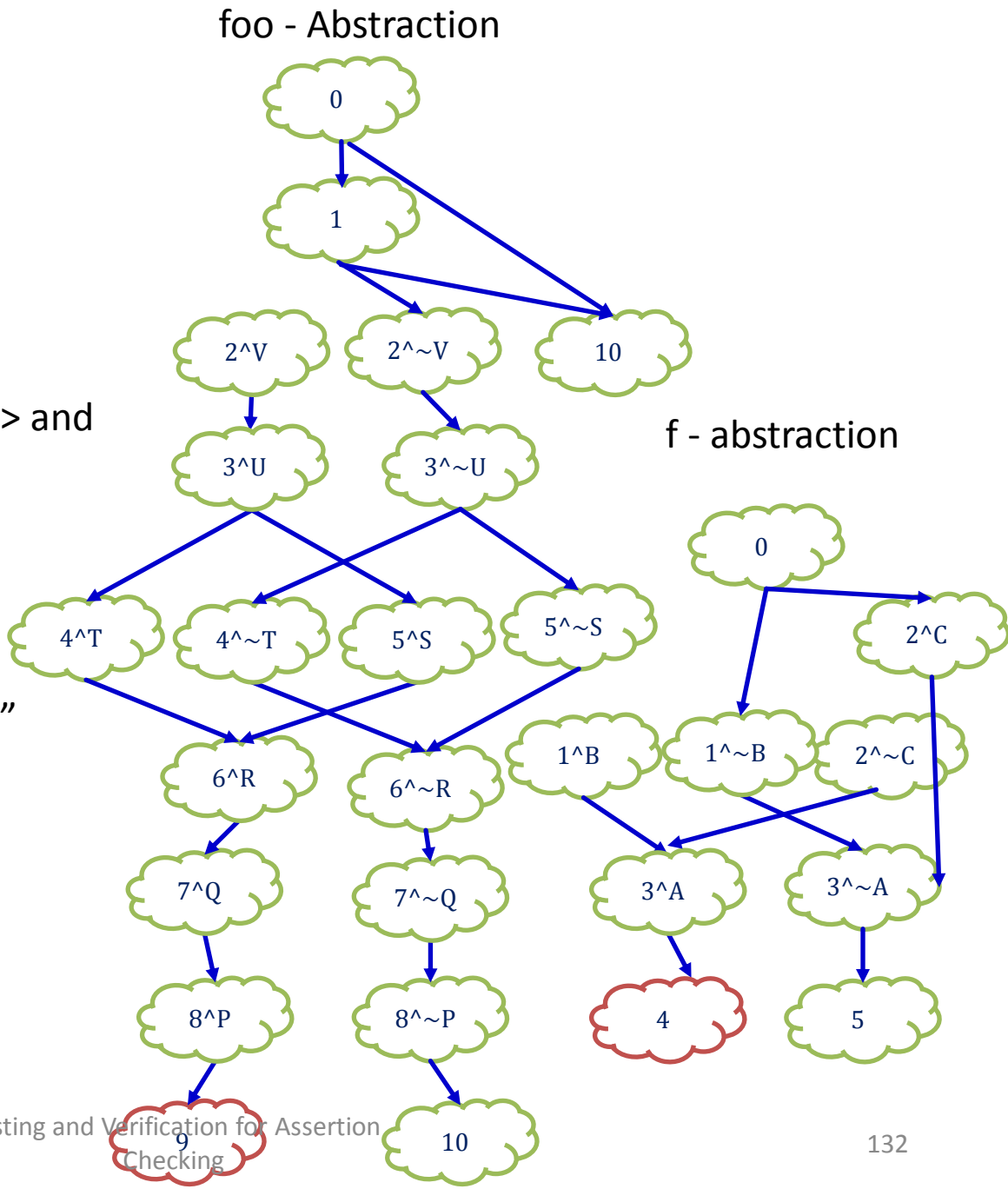
Step2: No proof

Step3: Abstract error trace is
<0, 1, 2, 3^U, 4^T, 6^R, 7^Q, 8^P, 9> and
Frontier is <2-(3^U)>

Step4: Inter procedure call

Calling DASH recursively with conditions on initial states as “true” and on final states as “ a < 0” (Since ret < 0)
Compute Refine prediction
= ~ (a < 0 | a > 0) = false = V

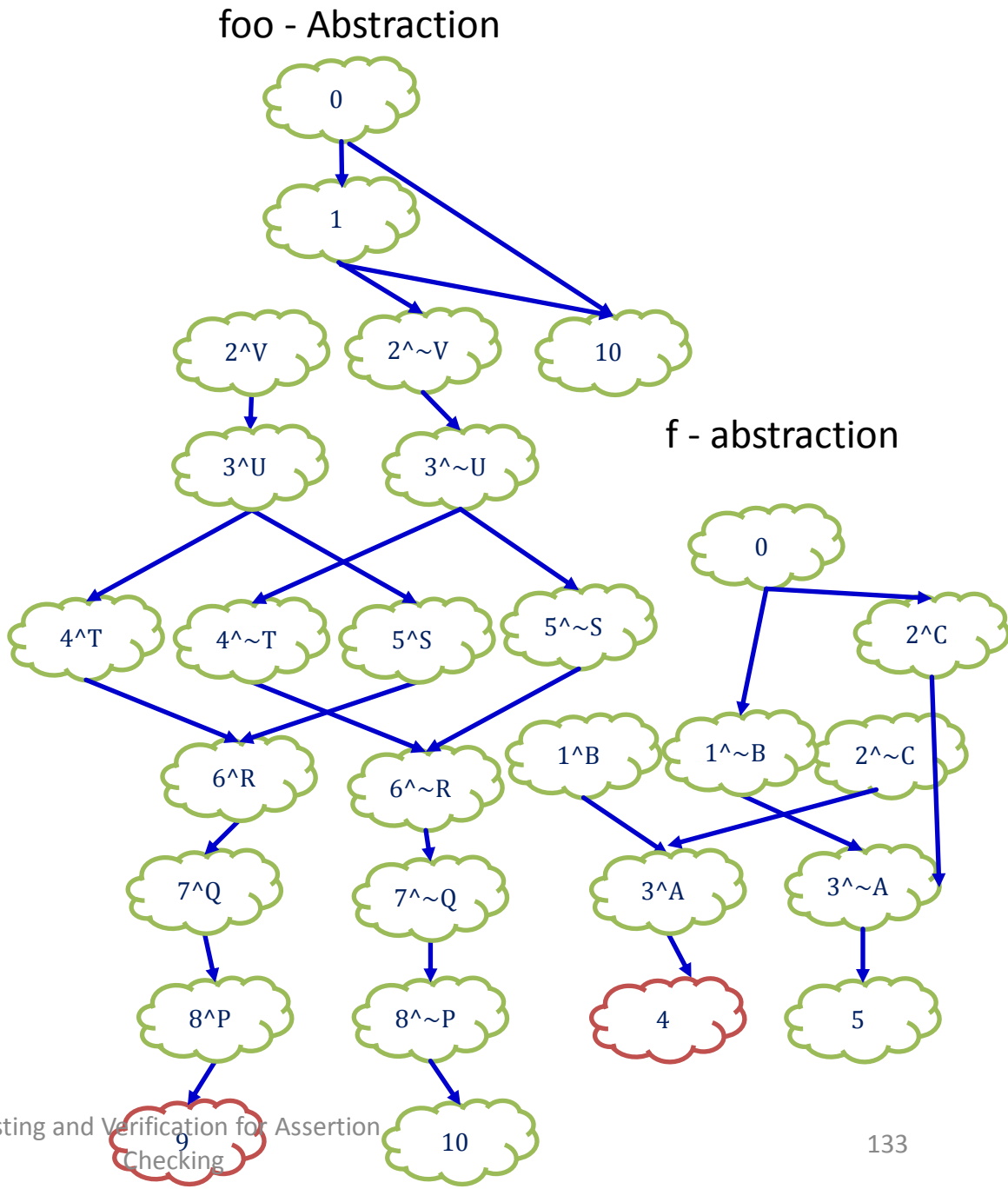
Splitting 2 into 2^V and 2^~V



Iteration – 9

Step1: No witness

Step2: Proof has been found out.



- If there is any function call inside a while loop of large number no of iterations, then DASH will be recursively invocated for that many number of times

- Time consuming

- Can we summarize the analysis and later reusing without analyzing once.

Outline

- Overview of Symbolic execution [DONE]
- Overview of (DART) [DONE]
- Synergy and its working on examples [DONE]
- DASH and its working on examples [DONE]
- SMASH and its working on examples
- Problem statement and Proposed solution
- Conclusion

Overview of SMASH

Extension to DASH by adding above ideas of summaries

- Assertion checking problem (Query) can be written as

$$\langle \varphi_{pre} \overset{?}{\Rightarrow}_f \varphi_{error} \rangle$$

  Where assertion is written in function f.

- Under approximate Analysis: Must Analysis:
  - Useful in finding bugs but not for proof
  - Proofs can be obtained after checking on all tests (Worst)

- Over approximate Analysis: May Analysis
  - Useful in giving proofs but not for tests
  - But can result in false errors

- SMASH unites both May and Must analysis compositionally.

- May analysis can be performed by
  - predicate abstraction
  - Counter example guided refinement.

- Must analysis can be performed by
  - Symbolic execution
  - Constraint solver

- SMASH is compositional
  - Save the analysis in form of summaries
  - Later will be reused.

May summary

- A may summary of a procedure P is of the form

$$\langle \varphi_1 \overset{may}{\Longrightarrow}_f \varphi_2 \rangle$$

  where φ_1 and φ_2 are predicates over program states.

- The may summary means that, if we invoke procedure f from any state satisfying φ_1 , the set of all possible states of the program on termination of P is over-approximated by the set of states φ_2.

- This implies that no states satisfying φ_2 are reachable from states satisfying φ_1 by executing P.

Not may summary

- A not may summary of a procedure P is of the form

$$\langle \varphi_1 \overset{\sim may}{\Longrightarrow}_f \varphi_2 \rangle$$

  where φ_1 and φ_2 are predicates over program states.

- The not may summary means that, for any state x $\in \varphi_1$, there does not exist a state y $\in \varphi_2$ such that the execution of f starting in state x terminates in state y.

- It clearly gives "NO" answer reply to query $\psi_1 \overset{?}{\Rightarrow}_f \psi_2$ if and only if $\psi_1 \in \varphi_1$ and $\psi_2 \in \varphi_2$

Must Summary

- A must summary of a procedure f is of the form

$$\langle \varphi_1 \overset{must}{\Longrightarrow}_f \varphi_2 \rangle$$

  where φ_1 and φ_2 are predicates over program states.

- The must summary means that, if we invoke procedure P from any state satisfying φ_1 , the set of all possible states of the program on termination of f is under approximated by the set of states φ_2.

Algorithm

When a query is given over a function, SMASH chooses one of the following actions.

1. If there exist a previously computed ¬May summary $\langle \hat{\varphi}_1 \overset{\neg May}{\Longrightarrow} P\hat{\varphi}_2 \rangle$ that answers the given query, then SMASH will use this ¬May summary directly without analyzing the procedure P.
2. If there exist a previously computed must summary $\langle \hat{\varphi}_1 \overset{must}{\Longrightarrow} P\hat{\varphi}_2 \rangle$ that answers the given query, then SMASH will use this must summary directly without analyzing the procedure P.
3. Otherwise, SMASH will analyze the procedure P and will make either ¬May summary or Must summary to answer the given query.

Must Summary

- A must summary $\varphi_1 \overset{must}{\Longrightarrow}_f \varphi_2$ implies that, for every state y $\in \varphi_2$, there exists a state x $\in \varphi_1$ such that the execution of f starting in state x terminates in state y.

- Must summary give "YES" answer to query query $\psi_1 \overset{?}{\Rightarrow}_f \psi_2$ if and only if $\varphi_1 \in \psi_1$ and $\psi_2 \cap \varphi_2$!= 0

Computing must summary

```
void main(int i1,i2) {
0: int x1,x2;
1: x1 = f(i1);
2: x2 = f(i2);
3: if (x1 > 0 & x2 > 0)
4:    error;
}
```

```
int g( int i) {
0:  if ( i > 0 )
1:     return i;
    else
2:     return -1;
}
```

Explain on board

Computing not may summary

```
void main(int i1,i2) {
0: int x1,x2;
1: x1 = g(i1);
2: x2 = g(i2);
3: if ((x1 < 0)||(x2 < 0))
4:   error;
5}
```

```
int g ( int i ) {
0:  if ( i > 0 )
1:    return i;
    else
2:    return -i;
 }
```

Explain on board

Computing must summary from not may summary

```
 void main ( int i ) {
0: int a = g(i);
1: if ( a == 0)
2:   error;
3:}
```

```
  int g (int i) {
 0:  if ( i > 0 )
1:     return h(i) + 1;
       else
 2:     return 0 ;
 3:}
```

$$\langle \text{true} \stackrel{\neg May}{\Longrightarrow}_h (\text{retval} = -1) \rangle$$

Explain on board

Computing not may summary from must summary

```
void main ( int i ) {
0: int a = g(i);
1: if ( a < 0)
2:    error;
3:}
```

```
int g (int i) {
0:  int a = h(i);
1:  if ( a < 10)
2:     return k(a);
    else
3:     return 0;}
```

```
int k ( int i) {
0:  if ( i > 0)
1:     return i;
     else
2:     return -i; }
```

$$\langle \mathrm{i} > 10 \overset{must}{\Longrightarrow}_h (15 < \mathrm{retval} < 20) \rangle$$

$$\langle \mathrm{i} < 14 \overset{must}{\Longrightarrow}_h (\mathrm{retval} < 11) \rangle$$

Explain on board

Outline

- Overview of Symbolic execution [DONE]
- Overview of (DART) [DONE]
- Synergy and its working on examples [DONE]
- DASH and its working on examples [DONE]
- SMASH and its working on examples [DONE]
- Problem statement and Proposed solution
- Conclusion

Problem statement and Solution

Limitation of SMASH algorithm over recursive programs

- It is as stated in paper that SMASH would not be guaranteed to terminate over recursive programs which have infinite data type domains or dynamic memory allocations.

- Consider the following recursive program which has infinite data type domain. SMASH results in non termination.

```
int main ( int x ) {
0:      int a = g(x);
1:      if  ( a < 0)
2:        error;
3:      return 1;
  }
```

```
int g (int x) {
0:     if ( x == 1 )
1:        return 1;
       else
2:        return g(x-1) + 1;
  }
```

Now, we consider another recursive program, which has finite data type domain and no dynamic memory allocation. But SMASH results in termination over this program.

```
int main ( int x ) {
0:      int a = g(x);
1:      if  ( a < 0)
2:        error;
3:      return 1;
  }
```

```
int g (int x) {
0:     if ( x < 0 or x > 3 ) \\ For finite domain
1:        return 0;
       else
2:        if ( x == 1)
3:            return 1;
          else
4:            return g(x-1) + 1;
}
```

Surprisingly, we consider another recursive program, which has finite data type domain and no dynamic memory allocation. But SMASH results in non termination over this program.

```
int f ( int x ) {
0:    int a = g(x);
1:    if ( a < 0 )
2:      error;
3:    return 1;
}
```

```
int g ( int x) {
0:   if ( x < 1 and x > 8 )  \\ For finite domain
1:      return 0;
     else
2:      if ( x == 1)
3:         return 1;
4:      else
5:         a = h(x);
6:         b = g(a-1);
7:         return b;
}
```

$$\langle \text{ x } = 5 \overset{must}{\Longrightarrow}_h (\text{ retval } = 6) \rangle$$

Problem

For a certain class of recursive programs, SMASH is generating queries continuously, that are answerable by in-progress queries. This is leading SMASH to be non terminating.

If it generates queries that are answerable by in-progress queries, then the sub sequent queries will be aroused from that query. It will keep on going and leading SMASH to be non terminating.

Solution

When a query is given over a function, SMASH chooses one of the following actions.

1. If there exist a previously computed ¬May summary $\langle \hat{\varphi}_1 \overset{\neg may}{\Longrightarrow} P\hat{\varphi}_2 \rangle$ that answers the given query, then SMASH will use this ¬May summary directly without analyzing the procedure P.

2. If there exist a previously computed must summary $\langle \hat{\varphi}_1 \overset{must}{\Longrightarrow} P\hat{\varphi}_2 \rangle$ that answers the given query, then SMASH will use this must summary directly without analyzing the procedure P.

Solution

3. If there exist no in-progress query that can answer the given query, then allow SMASH to analyze the procedure P and make either ¬May summary or Must summary to answer the given query.

4. If there exists an in-progress query that can answer the given query, then reply to that query by making query itself as ¬May summary. Then, the state before frontier gets refined and partitions the state into state following path leading to query that is answerable from any in-progress query and another state which possibly make a new query over the same function. In this case, it leads to following situations.

   (a) If it encounters an error and lands up with must summary, then it can be used directly to answer the query.

   (b) If it doesn't encounter an error and land up with ¬May summary, then we cannot say that there are no errors because we have willingly removed the case where SMASH leads to non termination by populating the query, that is answerable from in-progress queries, as ¬May summary.

Conclusion

- Just tested this idea on some problems(Pen and Paper)
- Would like to test this idea on real frame work in future.
- This thesis summarize the contributions of research work in the area of testing and verification for proving safety properties of programs.
- Since this thesis summarizes the algorithms for testing and verification, it helps a lot to students who are going to start work in this area.
- Described the algorithms in a more readable form and have ignored details of theoretical analysis of algorithms and practical implementations.

Outline

- Overview of Symbolic execution [DONE]
- Overview of (DART) [DONE]
- Synergy and its working on examples [DONE]
- DASH and its working on examples [DONE]
- SMASH and its working on examples [DONE]
- Problem statement and Proposed solution [DONE]
- Conclusion [DONE]

Any Questions ?

Thank you ☺