# Energy profiling of software:
## static analysis fundamentals

John Gallagher

Roskilde University

**ICT-Energy: Energy consumption in future ICT devices**

Summer School, Fiuggi, Italy, July 7-12, 2015

# Acknowledgements

The partners in the EU ENTRA project

Kerstin Eder and team

Pedro López García and team

Henk Muller and team

Roskilde team

# ICT-Energy

Application Software

System Software

Architecture

Devices

Physics

# Why worry about energy of *software*?

- Energy is consumed by hardware
- Hardware is getting more and more energy-efficient

- So why worry about energy-efficiency at the software level?

# Reason 1

- We take the application programmer's viewpoint
  - programmers don't know much about hardware
  - high-level languages hide the platform from the programmer

# Reason 1 - continued

Energy-inefficient software

Energy-efficient hardware

- Something like driving an energy-efficient car badly

# Reason 2

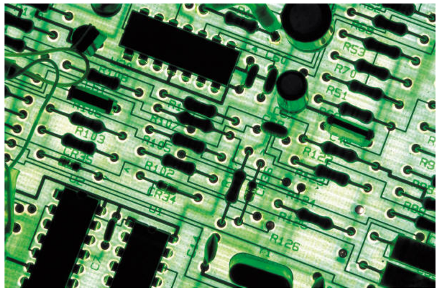
- Energy efficiency as a design goal from the start
- Get an energy profile for a program as early as possible
- Analyse the code to find out how much energy a program will use
- Deliver software with energy guarantees

# Reason 2 - continued

- Don't wait to test energy efficiency on hardware, after the software is developed

Development machine

Deployment platform



- It might be too late to fix "energy bugs"

# Reason 3

- You can save more energy at the software level than the hardware level

- There are more energy optimisation opportunities higher up the system stack.
- Most energy is wasted by application software

# Energy transparency

- Our aim is to let the programmer “see” the energy usage of the code
  - without executing it
  - so that the programmer can see where the program wastes energy
- In a similar spirit as UPPAAL formal models, directly on program code.

# Example

```
int biquadCascade(biquadState &state, int xn) {
    unsigned int ynl;
    int ynh;

    for(int j=0; j<BANKS; j++) {
        ynl = (1<<(FRACTIONALBITS-1));
        ynh = 0;
        {ynh, ynl} = macs( biquads[j].b0, xn, ynh, ynl);
        {ynh, ynl} = macs( biquads[j].b1, state.b[j].xn1, ynh, ynl);
        {ynh, ynl} = macs( biquads[j].b2, state.b[j].xn2, ynh, ynl);
        {ynh, ynl} = macs( biquads[j].a1, state.b[j+1].xn1, ynh, ynl);
        {ynh, ynl} = macs( biquads[j].a2, state.b[j+1].xn2, ynh, ynl);
        if (sext(ynh,FRACTIONALBITS) == ynh) {
            ynh = (ynh << (32-FRACTIONALBITS)) | (ynl >> FRACTIONALBITS);
        } else if (ynh < 0) {
            ynh = 0x80000000;
        } else {
            ynh = 0x7fffffff;
        }
        state.b[j].xn2 = state.b[j].xn1;
        state.b[j].xn1 = xn;

        xn = ynh;
    }
    state.b[BANKS].xn2 = state.b[BANKS].xn1;
    state.b[BANKS].xn1 = ynh;
    return xn;
```

**biquadCascade(BANKS) = 157 * BANKS + 51.7 nJoules**

This is an estimate of the energy used by the function.

It is a linear function of the value of BANKS



ENTRA

# Example

```
in port inP = XS1_PORT_4A;
out port led_port = XS1_PORT_1E;

void consumer(chanend couts) {
        int j;
        while (1) {
                couts :> j;
                for (int i=0;i<j;i++)
                        led_port <: (i & 1);
        }
}
```

12.3%

72.4%

Simulation with random 0..15 values on input port.

```
void producer(int n, chanend couts) {
        for (int i=0;i<n;i++) {
                printf("i=%d\n",i);
                couts <: i;
        }
}

int main () {
        chan a;  int x;
        par {
                while (1){
                        inP :> x;
                        producer(x,a);
                }
                consumer(a);
        }
}
```

13.8%

1.5%

# Energy a design goal for programmers

```
#pragma check energy(proc(x))<5pJ
int proc(int x) {
...
```

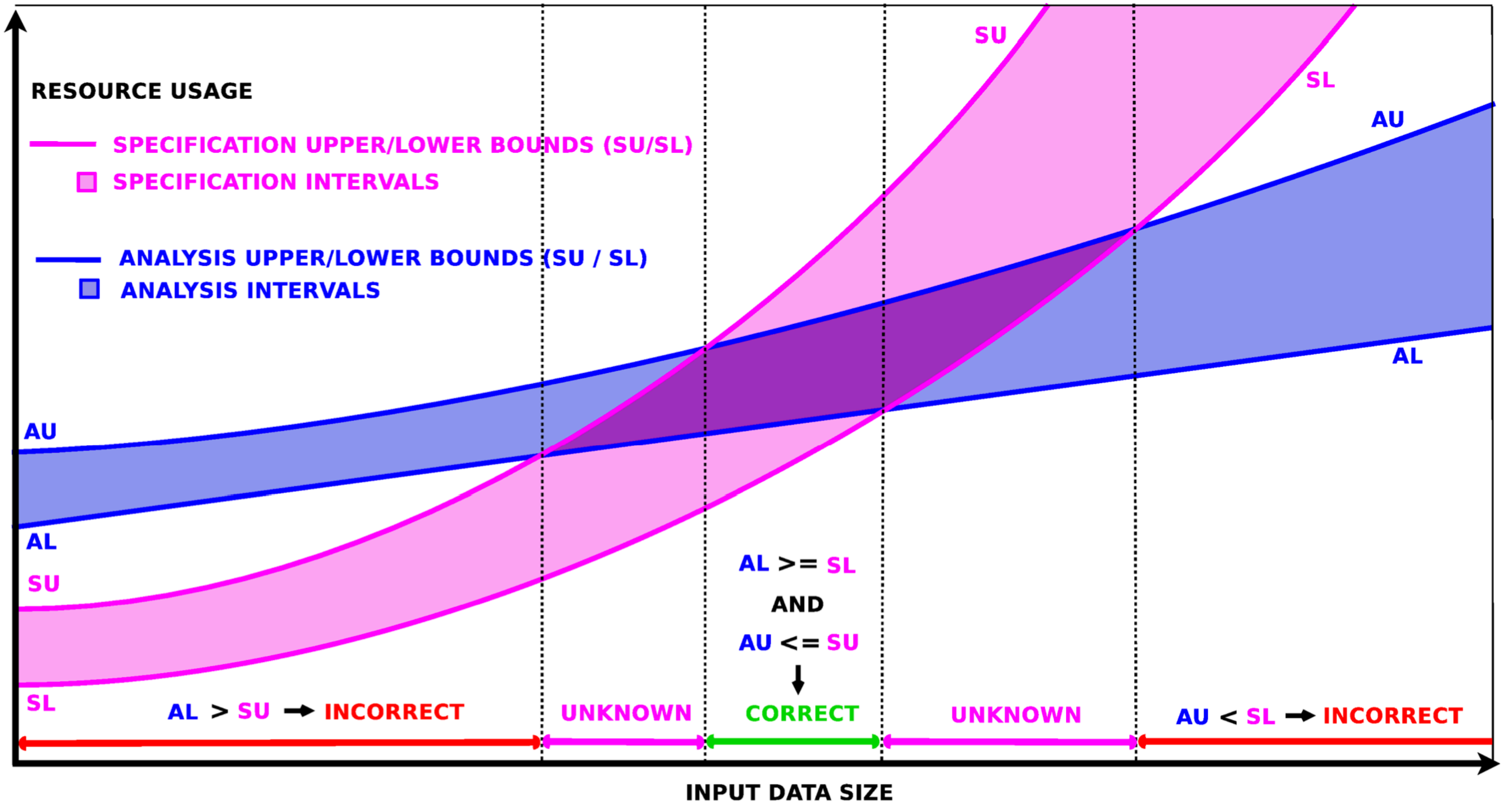
```
Output:

Checked 0≤x≤5⇒energy(proc(x))<5pJ
```



ENTRA

# Verification of energy specifications



# Summary of goals

- We want tools for the programmer
  - that give information about the energy usage of programs without running them (energy transparency)
  - that allow energy assertions to be checked (energy design goals)

# Analysis of programs

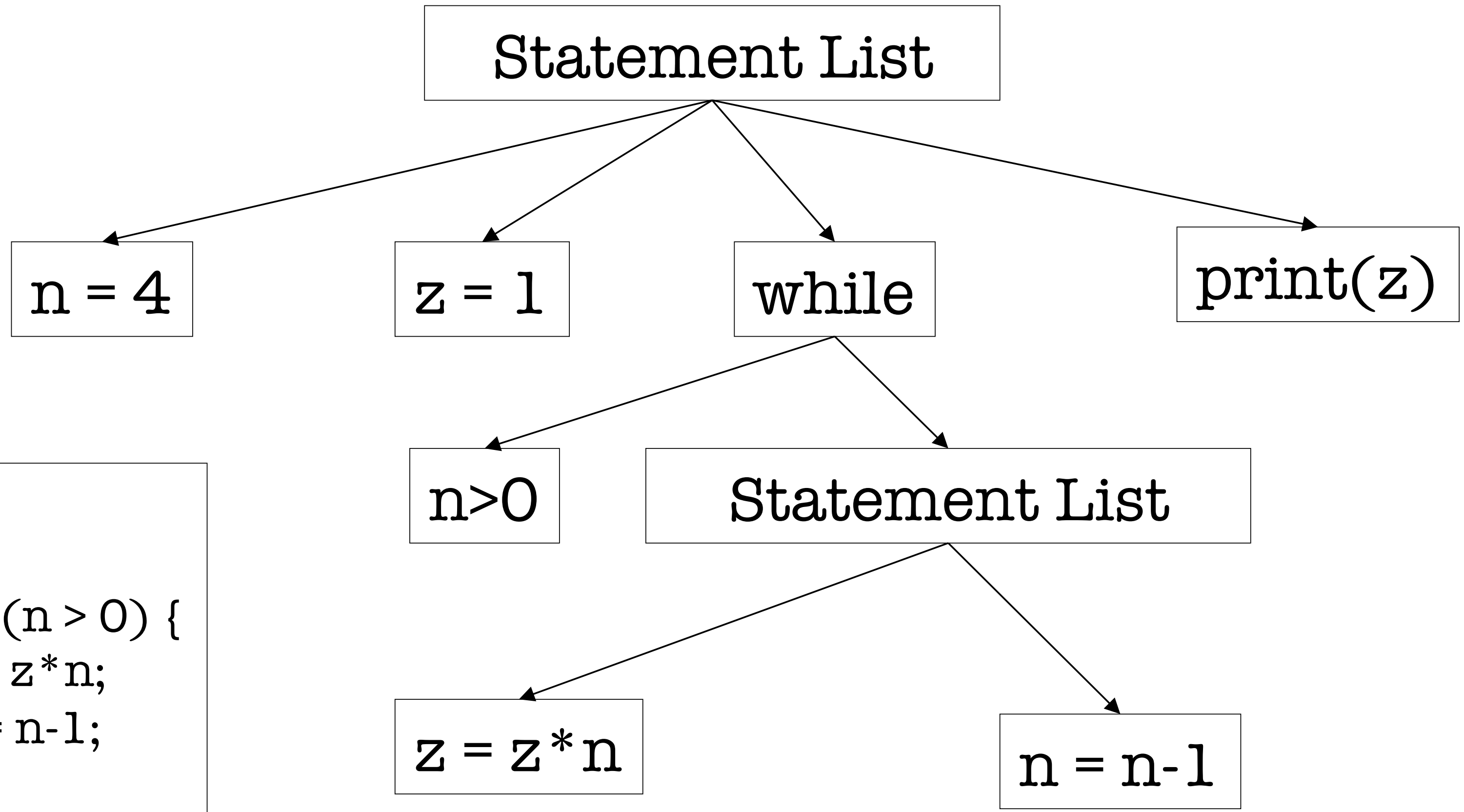
- A program is a physical object
  - some symbols on paper
  - a pattern of bits in memory
- But what is the meaning of a program?
- This is program semantics.

# Program semantics

```
n = 4;
z = 1;
while (n > 0) {
    z = z*n;
    n = n-1;
}
print(z);
```

To execute or analyse this program, we need to understand the meaning of "while", "semicolon", "{", "}", etc.

# Program syntax tree (parsing)




```
n = 4;
z = 1;
while (n > 0) {
    z = z*n;
    n = n-1;
}
print(z);
```

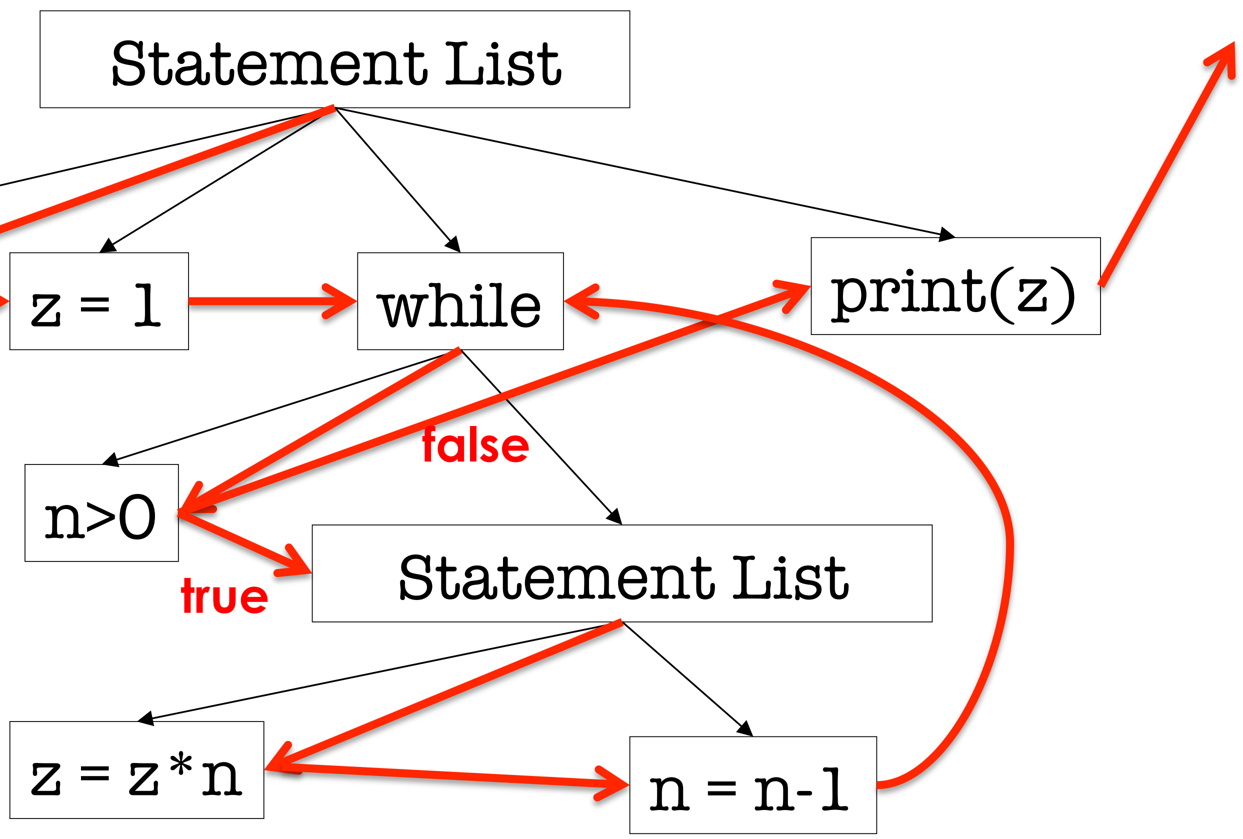
# From syntax tree to flow graph

| Grammar Rules | Semantic Rules for flow of control |
|---|---|
| If $\rightarrow$ if E then $S_1$ else $S_2$ | E.true := $S_1$ |
| | E.false := $S_2$ |
| | $S_1$.next := If.next |
| | $S_2$.next := If.next |
| While $\rightarrow$ while E $S_1$ | E.true := $S_1$ |
| | E.false := While.next |
| | $S_1$.next := While |
| StatementList $\rightarrow$ $S_1 S_2$ ..... $S_n$ | $S_j$.next = $S_{j+1}$ (j = 1 to n-1) |
| | $S_n$.next := StatementList.next |
| S $\rightarrow$ StatementList \| If \| While \| Print \| Assign | StatementList.next := S.next |
| | If.next := S.next |
| | While.next := S.next |
| | Print.next := S.next |
| | Assign.next := S.next |



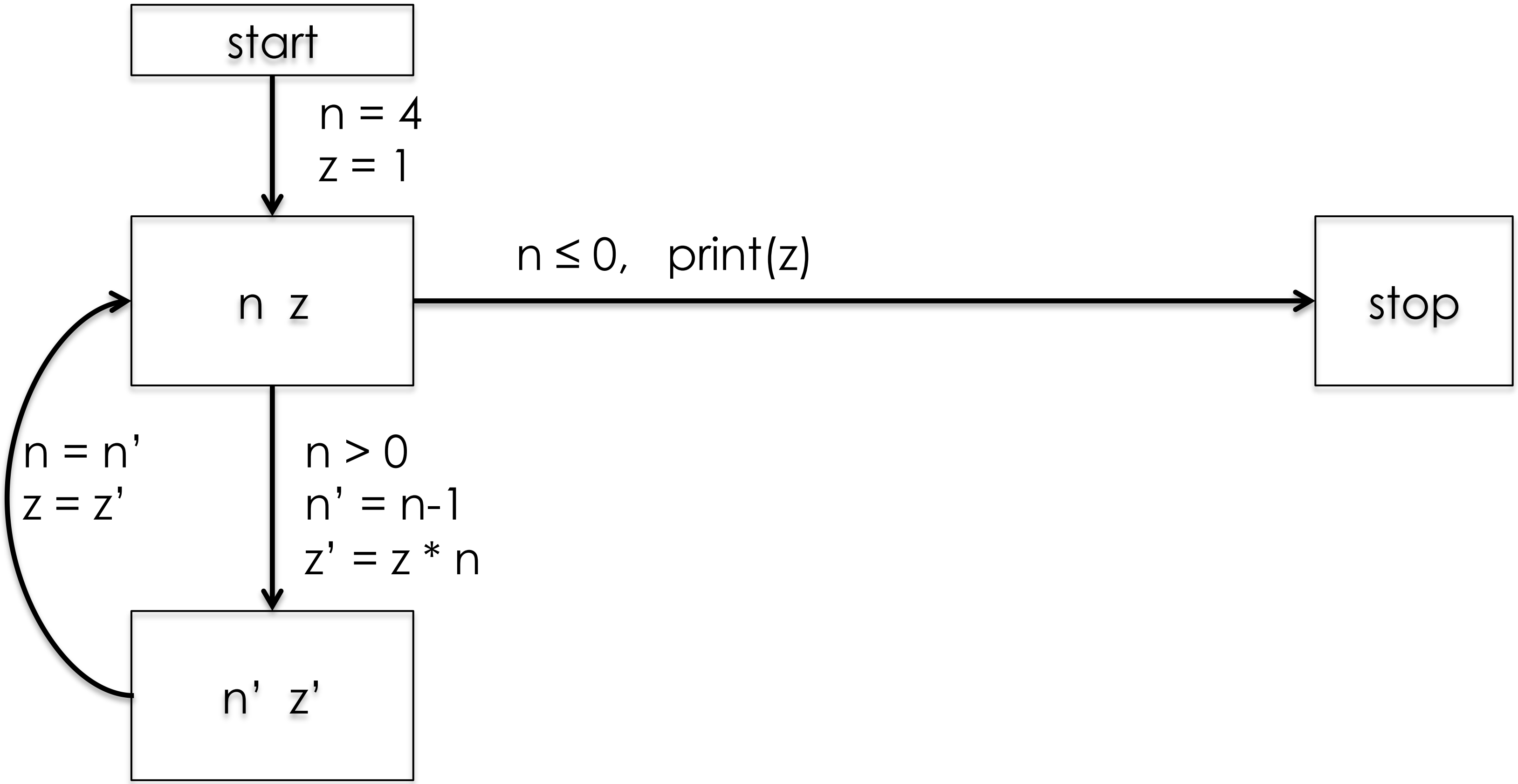
ENTRA

# From syntax tree to flow graph




```
n = 4;
z = 1;
while (n > 0) {
    z = z*n;
    n = n-1;
}
print(z);
```

# From flow graph to state automata



# From automaton to predicate logic

$$
\begin{aligned}
&\mathit{true} \rightarrow \quad \mathit{reachable}_1 \\
&(\mathit{reachable}_1 \wedge n=4 \wedge z=1) \\
&\qquad\qquad \rightarrow \quad \mathit{reachable}_2(n,z) \\
&(\mathit{reachable}_2(n,z) \;\; \wedge n<0 \wedge z'=z*n \wedge n'=n-1) \\
&\qquad\qquad \rightarrow \quad \mathit{reachable}_3(n',z') \\
&(\mathit{reachable}_3(n',z') \wedge n=n' \wedge z=z' \;) \\
&\qquad\qquad \rightarrow \quad \mathit{reachable}_2(n,z) \\
&\mathit{reachable}_2(n,z) \wedge n \geq 0 \wedge \mathit{print}(z) \;) \\
&\qquad\qquad \rightarrow \quad \mathit{stop}
\end{aligned}
$$

# Exercise

1. Draw the syntax tree

2. Draw the control flow graph

3. Draw the state automaton

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

# Logical representation

program point j — transition constraint — program point k

$$\boxed{x_1, x_2, \ldots, x_n} \xrightarrow{e(x_1, x_2, \ldots, x_n, x'_1, x'_2, \ldots, x'_n)} \boxed{x'_1, x'_2, \ldots, x'_n}$$

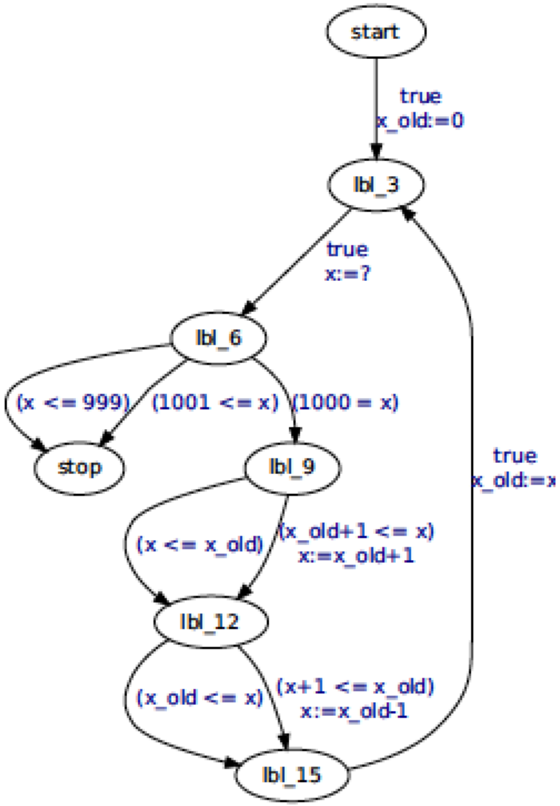
$$\boxed{\begin{array}{c}(reachable_j(x_1, x_2, \ldots, x_n) \wedge e(x_1, x_2, \ldots, x_n, x'_1, x'_2, \ldots, x'_n)) \\ \rightarrow \quad reachable_k(x'_1, x'_2, \ldots, x'_n)\end{array}}$$

# Example: A rate limiter*

*Example by Monniaux

**Listing 5.** Rate limiter

```
void main() {
  float x_old, x;
  x_old = 0;
  while (1) {
    x = input(-1000,1000);
    if (x >= x_old+1)
      x = x_old+1;
    if (x <= x_old-1)
      x = x_old-1;
    x_old = x;
  }
}
```



# Rate limiter – logic representation

```
r1(X,X_old) :-
        X_old=0,
        r0(_,_).
r1(X,X_old) :-
        r5(X,X_old).

r2(X,X_old) :-
        X >= -1000,
        X =< 1000,
        r1(_,X_old).

r3(X,X_old) :-
        X1 >= X_old+1,
        X = X_old+1,
        r2(X1,X_old).
```

```
r3(X,X_old) :-
        X < X_old+1,
        r2(X,X_old).

r4(X,X_old) :-
        X1 =< X_old-1,
        X = X_old-1,
        r3(X1,X_old).
r4(X,X_old) :-
        X > X_old-1,
        r3(X,X_old).

r5(X,X_old) :-
        X_old=X,
        r4(X,_).
```

# Invariants

- Many program analysis and verification tasks involve proving invariants
- An invariant is an assertion that is true at a given program point.

# Example invariant

```
void main() {
  float x_old, x;
  x_old = 0;
  while (1) {
    x = input(-1000,1000);
    if (x >= x_old+1)
        x = x_old+1;
    if (x <= x_old-1)
        x = x_old-1;
    x_old = x;
  }
}
```

Check assertion

$-1000 \leq x\_old \leq 1000$

# Proving invariants

- To prove that invariant P holds at program point j, prove the following implication

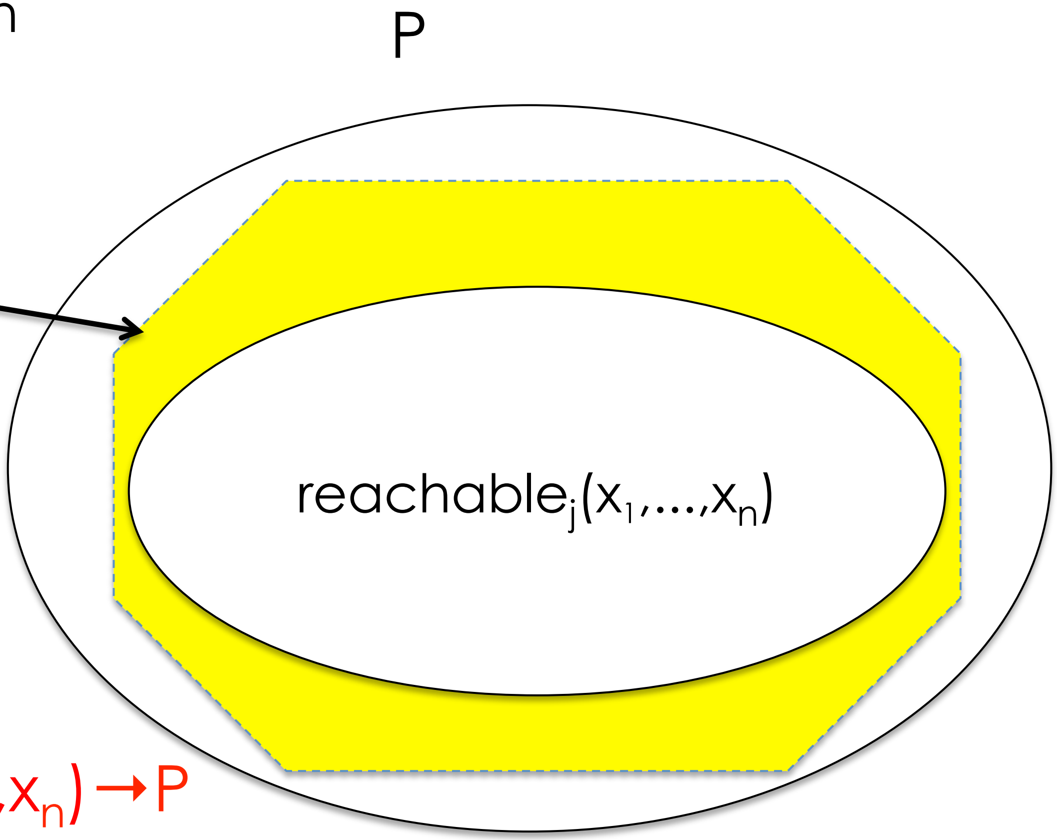
$reachable_j(x_1,\ldots,x_n) \rightarrow P$

which is equivalent to

$\neg(reachable_j(x_1,\ldots,x_n) \wedge \neg P)$

# Proof by approximation

Overapproximation of the set of points where $reachable_j(x_1,\ldots,x_n)$ is true.




Contained within P, hence

$reachable_j(x_1,\ldots,x_n) \rightarrow P$

# Energy invariants

- The program state can contain resource counters.
- $\text{reachable}_k(x_1,\ldots,x_n,e)$ means that the total energy consumed is $e$, when the program reaches point $k$

- So we can express and prove assertions about energy (or other resources)
- More on this later...

# Two basic techniques

- How to capture all reachable states?
  - answer, fixpoint techniques

- How to capture an infinite set of states?
  - answer, abstract interpretation

- These two methods underlie much program analysis


ENTRA

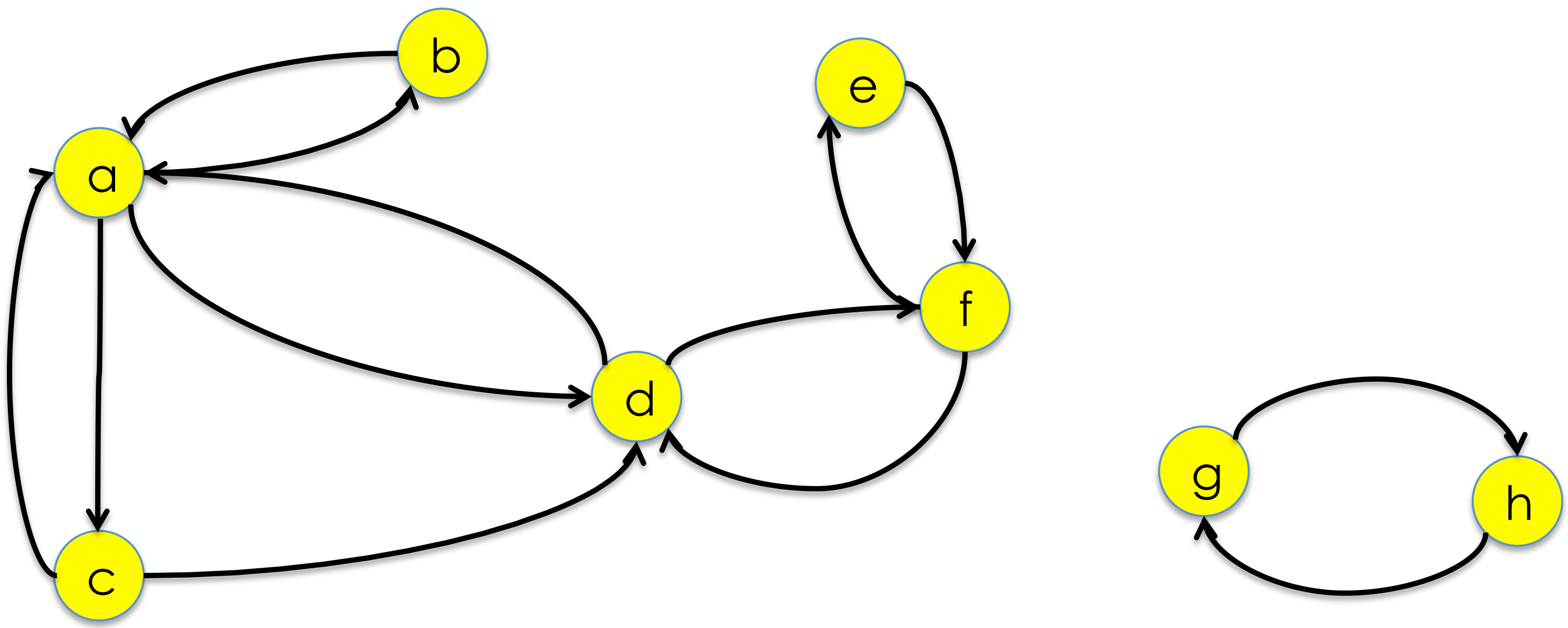
# Fixpoint computation

- Sounds complicated, but it is a very simple procedure

- It is a closure or saturation procedure

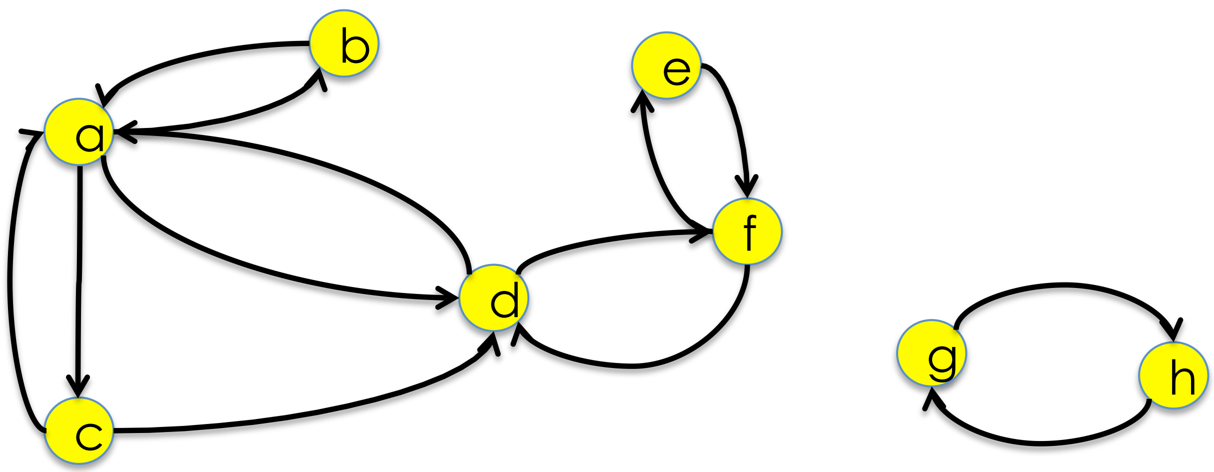
# Fixpoint example

- Consider a route network, with stations a,b,...,h



# post(S) function

- Let S be a set of stations. post(S) is the set of stations reachable in one step from S. E.g. post({a,h}) = {b,c,d,g}



# Reachability as a fixpoint

- The set of stations reachable from an initial set S, called Reach(S) is defined as the smallest set Z such that Z = F(Z)

  where F(Z) = S ∪ post(Z)

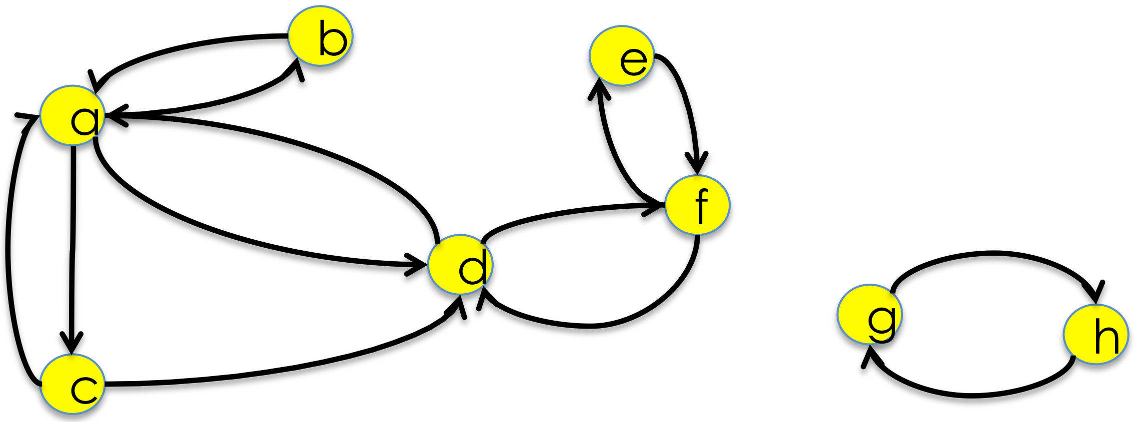
- This can be computed as the limit of a sequence $\emptyset$, F($\emptyset$), F(F($\emptyset$)), ...



ENTRA

# Example

- Find the stations reachable from a.




F(Z) = {a} ∪ post(Z)

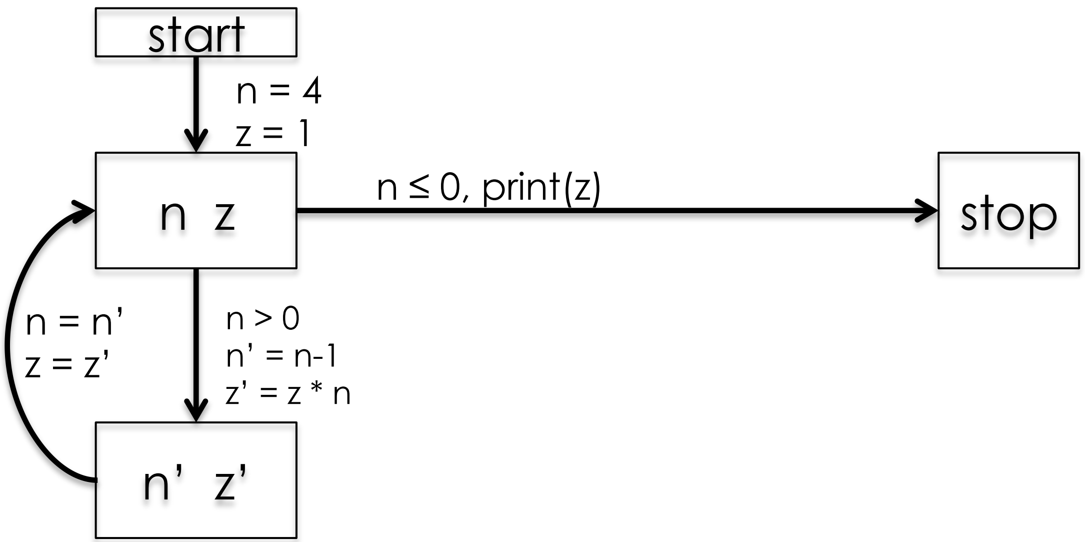
∅

F(∅) = {a}
F({a}) = {a,b,c,d}
F({a,b,c,d}) = {a,b,c,d,f}
F({a,b,c,d,f}) = {a,b,c,d,e,f}
F({a,b,c,d,e,f}) = {a,b,c,d,e,f}

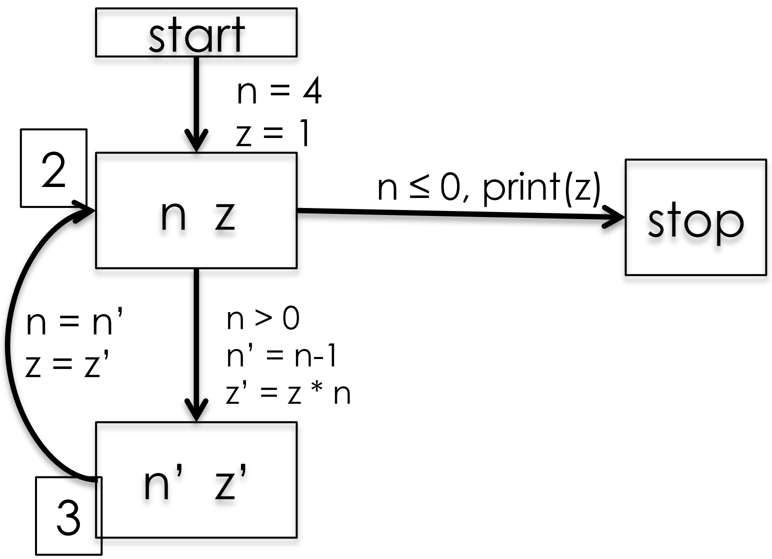
fixpoint found {a,b,c,d,e,f}

# The reachable states of a program

- We apply the same idea to find the reachable states of a program, starting with the initial state.



# The reachable states of a program




| 2 | 3 |
|---|---|
| {} | {} |
| {(4,1)} | {} |
| {(4,1)} | {(3,4)} |
| {(4,1),(3,4)} | {(3,4)} |
| {(4,1),(3,4)} | {(3,4),(2,12)} |
| …. | …. |
| {(4,1),(3,4),(2,12),(1,24),(0,24) } | {(3,4),(2,12),(1,24)} |

# Infinite fixpoints

- However, usually the set of reachable states of a program is infinite, and the sequence could keep on growing
- We might never reach the fixpoint
- In this case we use abstraction

# Abstract interpretation

Example

- $476305 \times -576 = 274351680$

- Is the above equation correct?

# Rule of signs

- The rule of signs is an abstraction of the multiplication relation

$$
\begin{aligned}
+ \times + &= + \\
+ \times - &= - \\
- \times + &= - \\
- \times - &= +
\end{aligned}
$$

We can check incorrectness, but not correctness with the rule of signs.

# The interval abstraction

- The value of a variable is abstracted by an interval
  - The variable has any value within the interval
- We can perform operations on intervals, as we did for signs

- E.g. [3,10] + [-2,6] = [3+(-2), 10+6] = [1,16]

- Exercise. What is [3,10] – [-2,6]?

# Example: interval abstraction

- The set of pairs of values {(4,1),(3,4), (2,12),(1,24),(0,24) } can be abstracted by the pair of intervals ([0,4], [1,24])
- So n is between 0 and 4, z is between 1 and 24.
- But information has been lost
  - the pair (3,19) is also consistent with the intervals.
  - the intervals give an over-approximation of the reachable states.



ENTRA

# Convex polyhedra

- A more precise abstraction than intervals is given by convex polyhedra

- Convex polyhedra are linear inequalities among the state variables

# Example convex polyhedron abstraction

```
var i,j:int;
begin
  i=0; j=10;
  while i<=j do
    i = i+2;
    j = j-1;
  done;
end
```

```
r1(I,J) :-
        I=0,J=10.
r2(I,J) :-
        r1(I,J).
r2(I,J) :-
        I1 =< J1,
        I = I1+2,
        J = J1-1,
        r2(I1,J1).
r3(I,J) :-
        I >= J+1,
        r2(I,J).
```

# Approximate reachable states

```
r1(I,J) = [I=0,J=10].
r2(I,J) = [-I >= -16,I >= 0,I+2*J=20].
r3(I,J) = [-3*I >= -26,3*I >= 22,I+2*J=20].
```

This result is computed fast, using the Parma Polyhedra Library to perform the operations on convex polyhedra.

# Summary so far....

- We can translate a program to a state automaton
- We can compute over-approximation of the reachable states of the program
  - using fixpoint computation and abstraction
- We can use the approximation to check assertions about the program.