Computing Time as a Program Variable:

a way around infeasible paths

Niklas Holsti

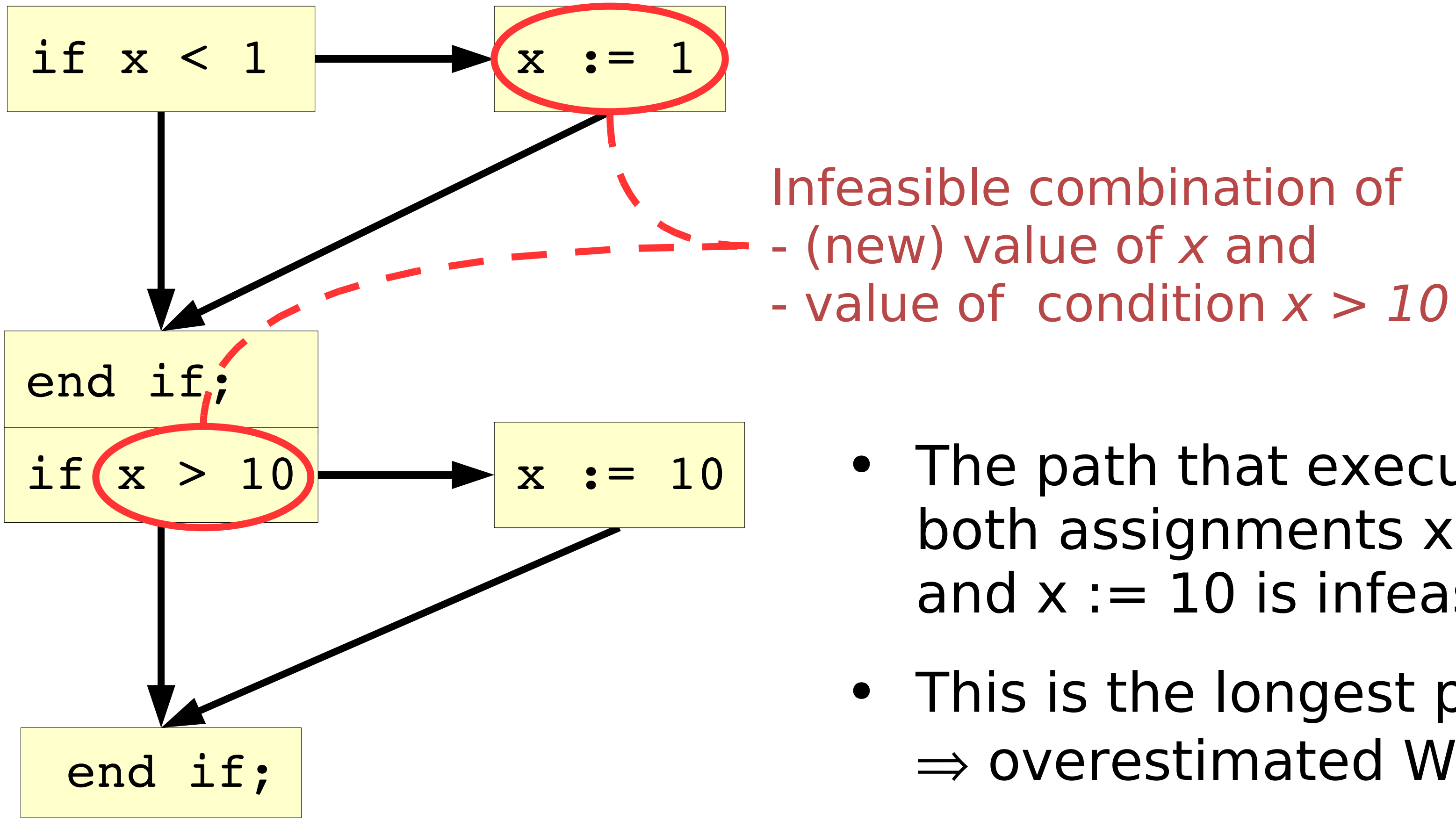
Tidorum Ltd

www.tidorum.fi

Example: Saturating a value

- Make sure that *x* is between 1 and 10:

```
if x <  1 then x :=  1; end if;
if x > 10 then x := 10; end if;
```

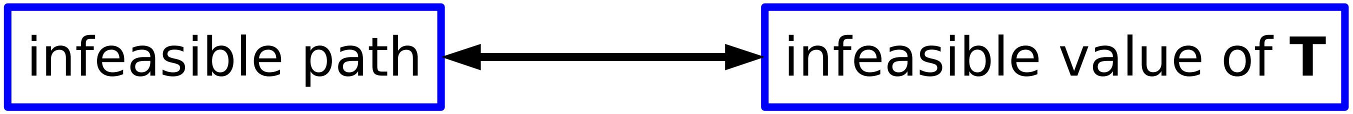


Infeasible combination of
- (new) value of *x* and
- value of condition *x > 10*

- The path that executes both assignments x := 1 and x := 10 is infeasible
- This is the longest path ⇒ overestimated WCET

Reasoning

- Infeasible paths arise from dependencies (correlations) between variable values/assignments and values of branch conditions
- Some analysis of such dependencies is necessary
- Earlier work: “path-oriented” analysis
  - find (in-) feasible combinations of CFG blocks/edges
  - use “flow facts” to constrain eg. IPET
- Suggestion: “value-oriented” analysis
  - find (in-) feasible combinations of variable values
  - make execution time a variable, **T**
  - thus, find (in-) feasible execution times = values of **T**

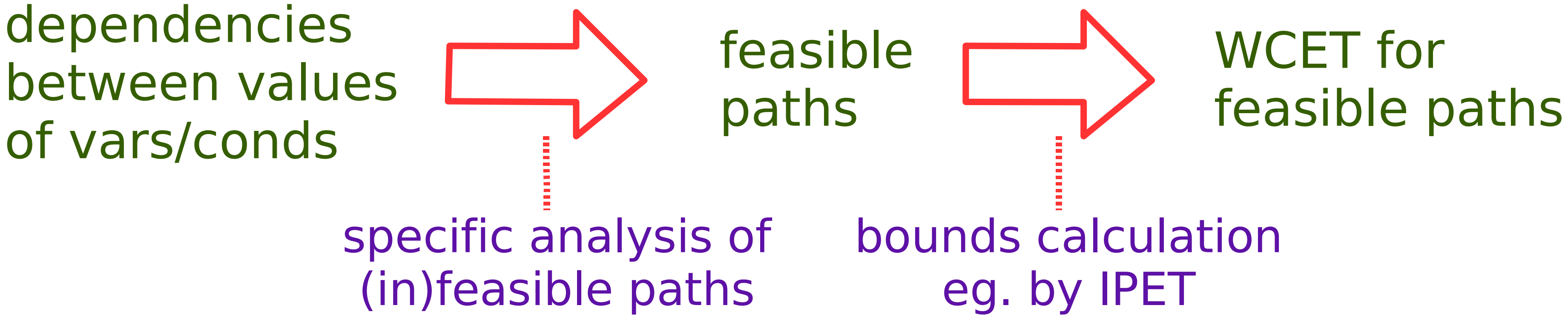


infeasible path

infeasible value of **T**

From values to feasible WCET

- Path-oriented:



dependencies between values of vars/conds

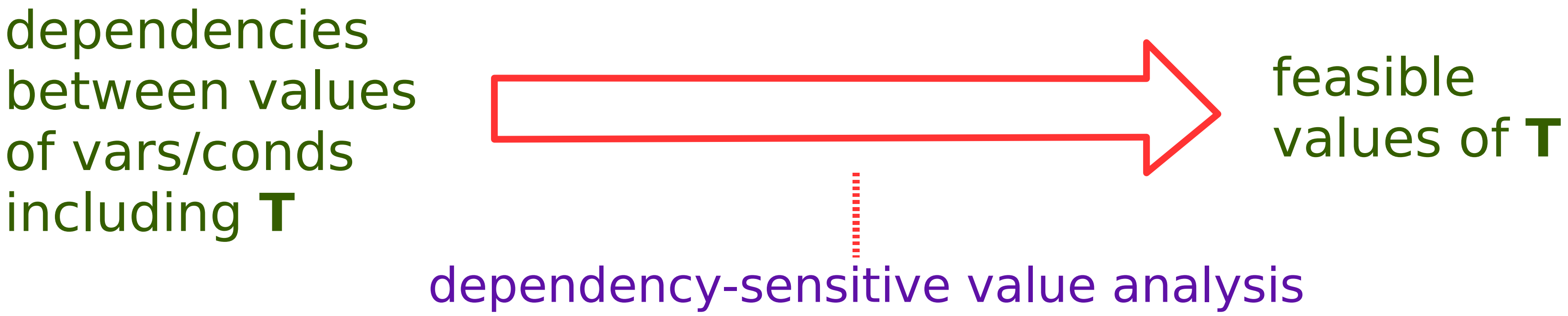
feasible paths

WCET for feasible paths

specific analysis of (in)feasible paths

bounds calculation eg. by IPET

- Value-oriented:



dependencies between values of vars/conds including **T**

feasible values of **T**

dependency-sensitive value analysis

Adding the T variable (in a real tool)

- Add **T** to the internal representation (CFG) of the machine-code program
- Add $\mathbf{T} := 0$ at the start of the program/subprogram
- Add $\mathbf{T} := \mathbf{T} + t(b)$ in each basic block b
  - $t(b)$ comes from processor-behaviour analysis
    - using all structural paths in the CFG
    - may include infeasible paths
- Ditto for control-flow edges that take some time
- Use interval arithmetic if $t(b)$ is not a single value
  - eg. context-dependent pipeline or cache effects

Adding T to the “saturation” example

- Add **T** as a variable in the (pseudo-) source code
- Add **else**-parts to model the condition-evaluation time.

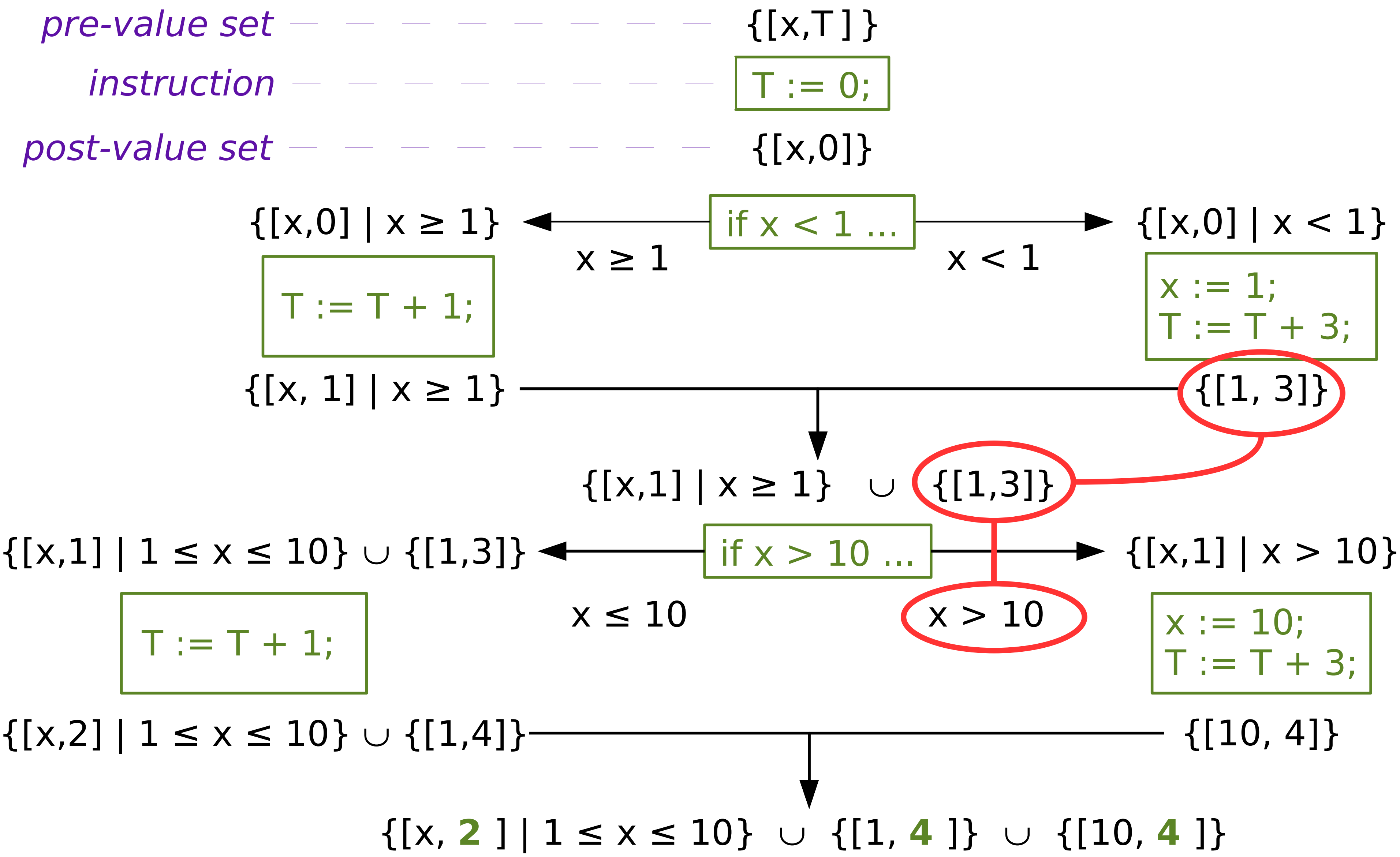
```
T := 0;

if x <  1 then x :=  1; T := T + 3;
          else          T := T + 1; end if;

if x > 10 then x := 10; T := T + 3;
          else          T := T + 1; end if;
```

- ET of program = final value of **T**
  - Infeasible path ET is 3 + 3 = 6 cycles.
  - WCET is 1 + 3 = 4 cycles.
  - BCET is 1 + 1 = 2 cycles.

A dependency-sensitive value-analysis

- This is just one method/domain; others are possible
  - similar to the analysis in the Bound-T WCET tool,
  - which Bound-T currently uses mainly for loop bounds
  - implemented with the Omega Calculator (Pugh et al.)
- Models:
  - value of one variable : integer $\in \mathbb{Z}$
  - combined values of *n* variables : *n*-tuple $\in \mathbb{Z}^n$
  - all combined values of *n* variables : *n*-tuple set $\subseteq \mathbb{Z}^n$
  - instruction : transfer relation $\subseteq \mathbb{Z}^n \times \mathbb{Z}^n = \mathbb{Z}^{2n}$
- Set : $\{ [v_1, v_2, \ldots, v_n] \mid \textit{constraints} \}$
- Relation : $\{ [v_1, v_2, \ldots, v_n] \rightarrow [v'_1, v'_2, \ldots, v'_n] \mid \textit{constraints} \}$
- Constraints in Presburger Arithmetic form: Presburger sets

Presburger-set analysis of the example



pre-value set — $\{[x,T]\}$

instruction — `T := 0;`

post-value set — $\{[x,0]\}$

`if x < 1 ...`

$x \geq 1$: $\{[x,0] \mid x \geq 1\}$

`T := T + 1;`

$\{[x, 1] \mid x \geq 1\}$

$x < 1$: $\{[x,0] \mid x < 1\}$

`x := 1;`
`T := T + 3;`

$\{[1, 3]\}$

$\{[x,1] \mid x \geq 1\} \cup \{[1,3]\}$

`if x > 10 ...`

$x \leq 10$: $\{[x,1] \mid 1 \leq x \leq 10\} \cup \{[1,3]\}$

`T := T + 1;`

$\{[x,2] \mid 1 \leq x \leq 10\} \cup \{[1,4]\}$

$x > 10$: $\{[x,1] \mid x > 10\}$

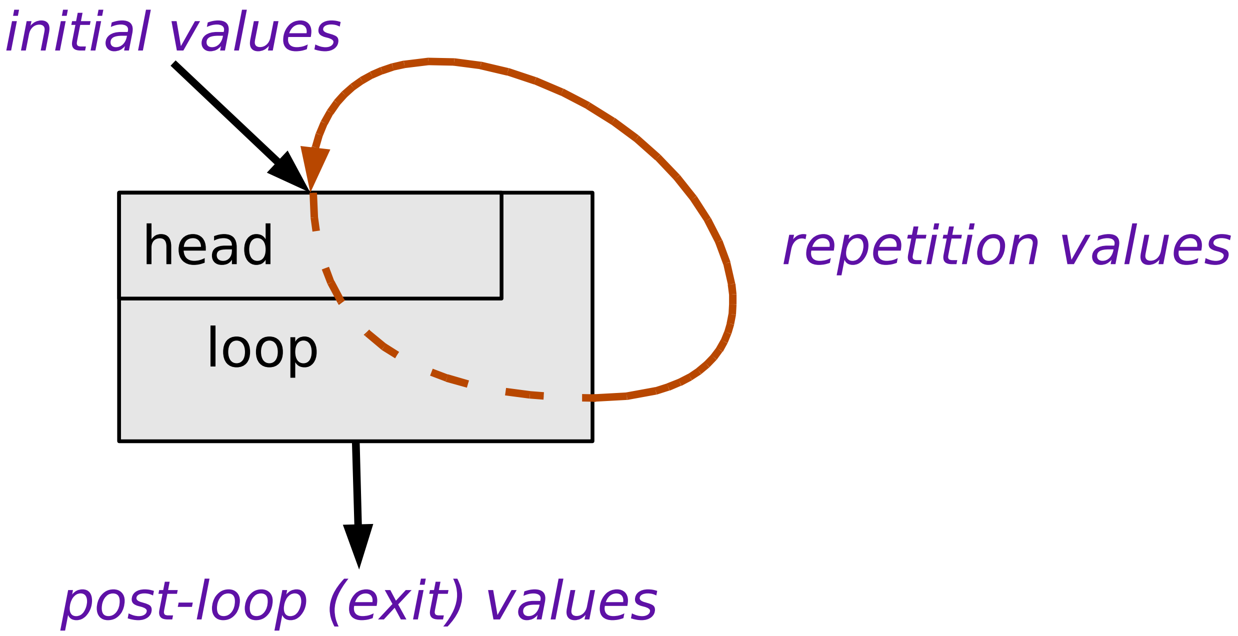
`x := 10;`
`T := T + 3;`

$\{[10, 4]\}$

$\{[x, \mathbf{2}\,] \mid 1 \leq x \leq 10\} \cup \{[1, \mathbf{4}\,]\} \cup \{[10, \mathbf{4}\,]\}$

What about loops?

- Three different things:
  - finding loop bounds (bounds on # of iterations)
  - finding the effect of loops on variable values
  - handling infeasible paths involving loops.
- Presburger-set analysis can be used for all three things
- Focus: how **T** -variable works in infeasible looping paths



The repetition relation of a loop

- The repetition relation of a loop shows how variable values change in one repetition of the loop
  - from the transfer relations of the instructions in the loop
  - exit from loop is treated separately (as normal flow)
- Example: Reverse order of *vec*[n .. $n + 9$]:

```
i := n; j := n + 9;
while i < j loop
   z := vec[i]; vec[i] := vec[j]; vec[j] := z;
   i := i + 1;
   j := j – 1;
end loop;
```

- Ignore the *vec*[] values (pointer analysis...)
- The repetition relation R for i, j, n, z is:

$$R = \{ [\, i, j, n, z \,] \rightarrow [\, i + 1,\ j - 1,\ n,\ z' \,] \ \mid\ i < j \}$$

Invariant, induction, and fuzzy variables

- Use the repetition relation *R* to classify each variable *v* as:
  - invariant: *R* does not change *v*
  - induction: *R* sets *v* := *v* + *dv*, where *dv* is constant $\in \mathbb{Z} \backslash \{0\}$
  - fuzzy: *R* changes *v* in other (unknown) ways
- Computation: see paper
  - intersect *R* with { [v, dv] → [v + dv] }
  - project to *dv*
  - take convex hull ⇒ interval of possible *dv*
- Example above:
  - *i* and *j* are induction variables; *di* = 1, *dj* = -1
  - *n* is invariant; in other words *dn* = 0
  - *z* is fuzzy.

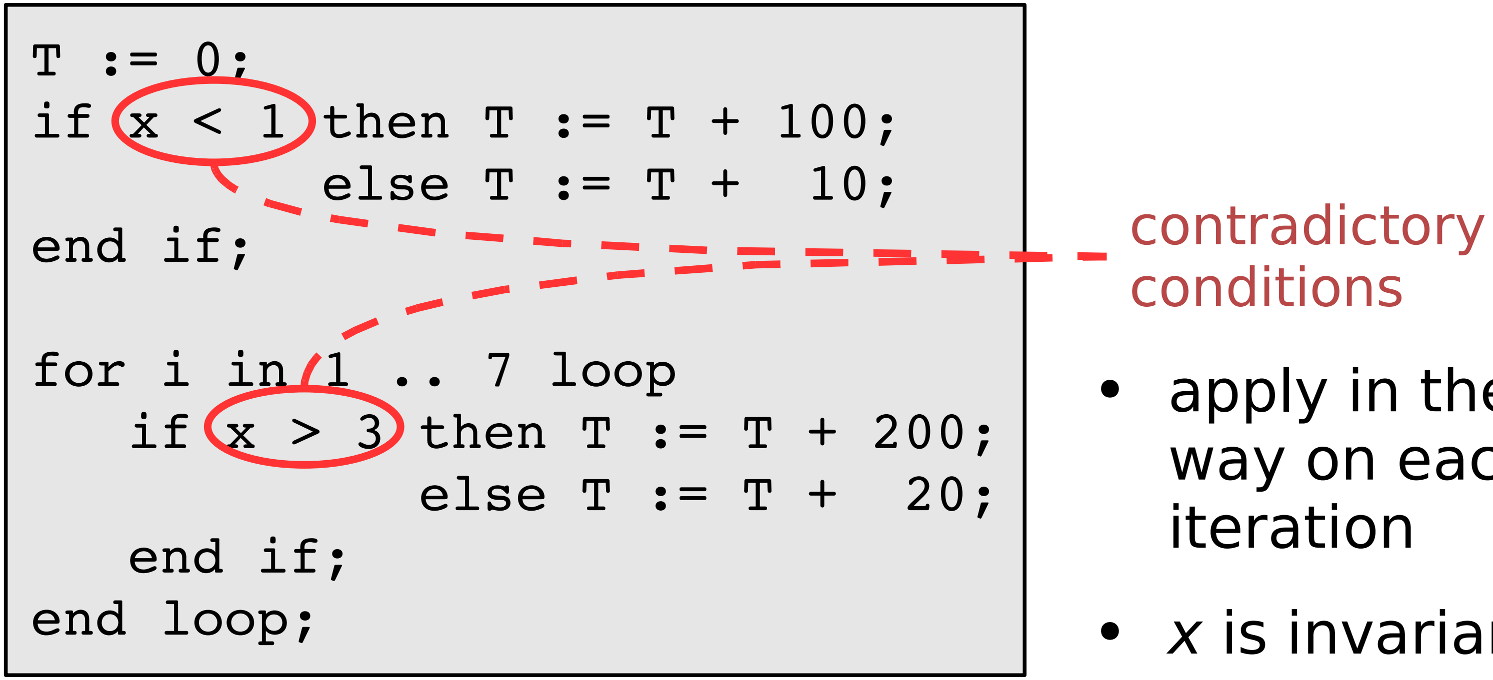
Induction model of loop

- Add iteration counter $c = 0, 1, \ldots$
- Induction model: Value of v at start of iteration c is
  - if invariant: $v = v_0$ = initial value of v
  - if induction: $v = v_0 + c \times dv$
  - if fuzzy: v is unconstrained (unknown)
- Loop bound N: see paper
  - propagate the induction model to the back edge:
    $\{ [i, j, n, z] \mid i - 1 < j+1 \text{ and } i = n+1+c \text{ and } j = n+8+c \}$
  - project to c
  - take convex hull ⇒ bounds on c that allow repetition
  - example: $c \leq 4$.
- Post-loop values: Propagate induction model to exit
  - effect on induction variable: $v := v + N \times dv$, N constant
    - plus possible effect of the loop-exit path

T is an induction variable

- Effect of loop is **T** := **T** + $N \times$ dT
  - dT is the execution time of one loop repetition
  - *N* is a constant (range)
- dT from dependency-sensitive analysis of loop body
  - excludes infeasible paths contained in the loop body
    - when infeasibility is iteration-independent
- dT is a Presburger variable
  - can depend on other variables
  - shows dependency between paths inside and outside loop
  - final **T** excludes infeasible combinations of such paths
    - when infeasibility is iteration-independent
- Hard to handle iteration-specific infeasibility
- Fails for infeasible “path – loop-bound” combinations

Example iteration-independent case



```
T := 0;
if x < 1 then T := T + 100;
         else T := T +  10;
end if;

for i in 1 .. 7 loop
   if x > 3 then T := T + 200;
            else T := T +  20;
   end if;
end loop;
```

contradictory conditions

- apply in the same way on each loop iteration
- *x* is invariant

- No path can take the slow case of both branches
  - infeasible longest path = 100 + 7 × 200 = 1500 cycles
  - longest feasible path = 10 + 7 × 200 = 1410 cycles
- **T**-analysis works; final **T** = 1410.

Summary

- Add execution-time variable **T**
  - **T** becomes “entangled” with other variables/conditions
- Use dependency-sensitive value-analysis
  - final value of **T** is “entangled” with feasible paths
- Good:
  - no specific analysis and representation of infeasible paths
  - handles many kinds of infeasible paths
- Bad:
  - history-sensitive $t(b)$ may be over-estimated
  - Presburger-set analysis is costly; hard to scale up
- Possible future work:
  - implementation and evaluation
  - other, cheaper dependency-sensitive value-analyses
    - but non-convexity (disjunction) is probably desirable