Execution Time Analysis and Abstract Interpretation

Björn Lisper
Dept. of Computer Science and Electronics
Mälardalen University

bjorn.lisper@mdh.se
http://www.idt.mdh.se/~blr/

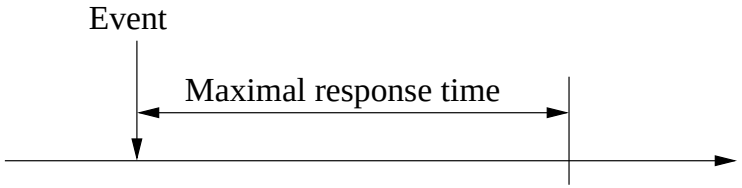
October 13, 2005

Outline

- Real-Time Systems
- Worst-Case Execution Time (WCET) Analysis
- Abstract Interpretation
- Abstract Interpretation Applied to WCET Analysis

Real-Time Systems

Real-Time Systems are systems with *timing constraints*



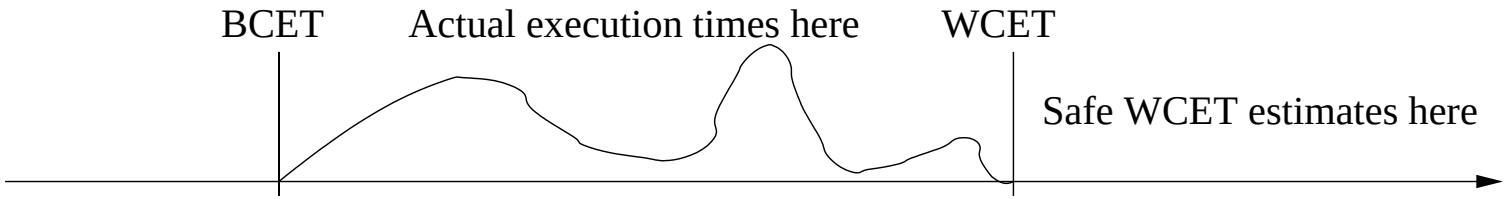
Absolute deadlines, maximum response times, bounds on execution time variation, etc.

Two classes:

- *Hard* real-time systems: timing constraints *must* be met (typically safety-critical systems)
- *Soft* real-time systems: *desirable* that timing constraints are met (but not absolutely necessary)

WCET

To guarantee timing constraints, we need to know the *Worst-Case Execution Time* (WCET) of pieces of code



Common practice is to measure, and add a safety margin

But this gives no strict guarantees! Often, exhaustive testing is not feasible

A formal analysis is a safer alternative!

WCET Analysis

Basic assumptions:

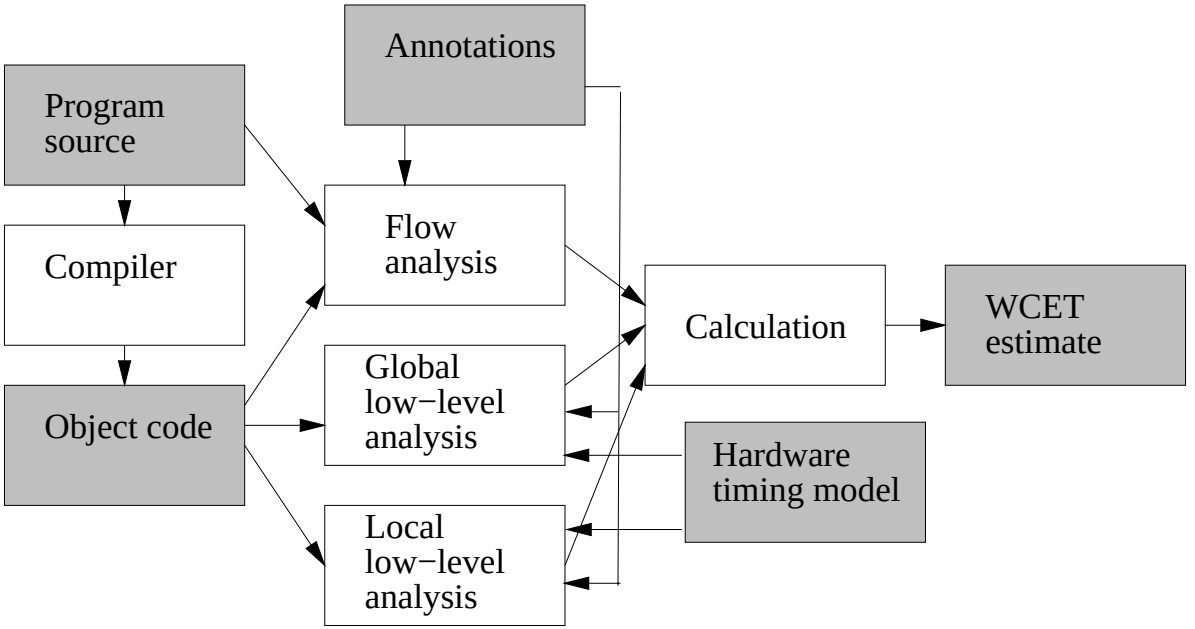
- sequential program
- the program runs in isolation (no interrupts)

WCET analysis must then:

- constrain possible program flows (“high-level”, or “flow” analysis)
- Estimate hardware impact on execution time for program fragments (“low-level” analysis)
- Use information to produce a safe WCET estimate (calculation)

The object (low-level) code is analyzed, not the source code

Structure of WCET analysis



Problems to Overcome

Exact WCET estimation is undecidable (would solve the halting problem)

Go for methods that give good enough results on many enough interesting programs. Fortunately, hard RT software tends to have simple structure

Modern CPU:s have very complex performance models

Much research has gone into how to handle this. Also, many embedded processors are very simple

In principle, the **low-level code** must be analyzed w.r.t. possible program flows. This is hard

Use info from higher level representations whenever possible. Also, more can be done with binary code than seems possible at first sight

Calculation

Three major techniques:

- *Tree-based* calculation
- *Path-based* calculation
- *The Implicit Path Enumeration Technique* (IPET)

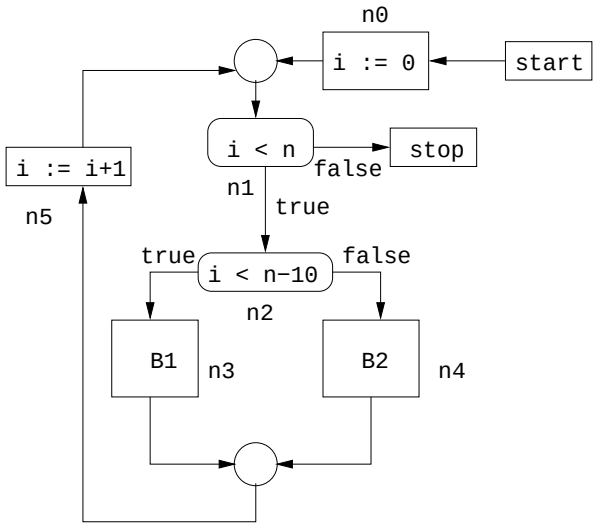
We’ll focus on IPET here

The Implicit Path Enumeration Technique

A constraint-based approach

Can deal with unstructured programs and nonlocal flow constraints

Based on control-flow graph (CFG) program representation:



For each node n_i in the CFG, obtain its computation time t_i by low-level analysis

Define execution counter x_i for each n_i

Flow constraints expressed as constraints on execution counters:

- $x_i \geq 0$ for all n_i (counters must be non-negative)
- Structural flow constraints: for each node, $\sum$ input counters $= \sum$ output counters
- $x_i = 1$ if n_i is start or stop node
- Loop iteration bounds: $x_i \leq n$
- Infeasible paths (mutual exclusivity): $x_i + x_j \leq 1$

WCET calculation then becomes a maximization problem:

$$\max \sum x_i t_i$$

subject to flow constraints

ILP problem if flow constraints are linear

Can be solved by standard methods

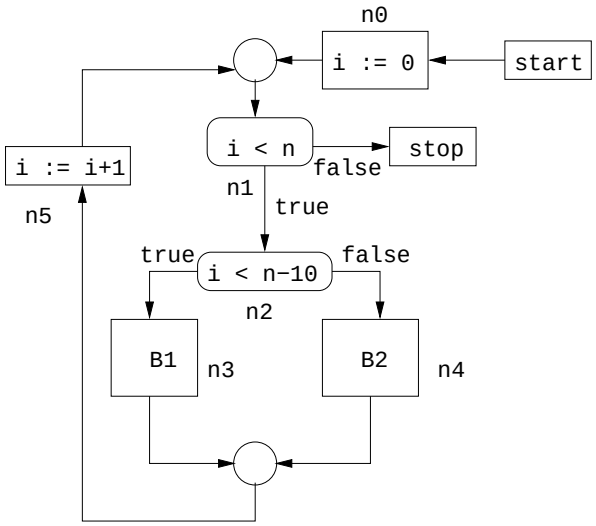
This model also assumes that timing is additive (order of execution does not matter, only number of executions)

There are methods to compensate for this, when the hardware has a non-additive timing model

Example

With structural constraints only (counters $x_0, \ldots, x_5$, assume execution times $t_0 = t_5 = 10$, $t_1 = t_2 = 5$, $t_3 = 50$, $t_4 = 100$):

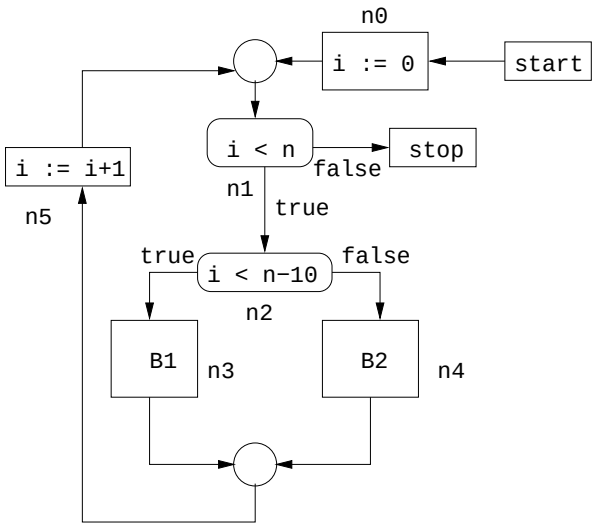
$$
\begin{aligned}
x_i &\geq 0, \quad i = 0, \ldots, 5 \\
x_0 &= 1 \\
x_1 &= x_0 + x_5 \\
x_2 &= x_1 - 1 \\
x_2 &= x_3 + x_4 \\
x_5 &= x_3 + x_4
\end{aligned}
$$



Unbound problem, no solution

Say we know $n = 20$. Add loop bound constraint $x_1 \leq 21$:

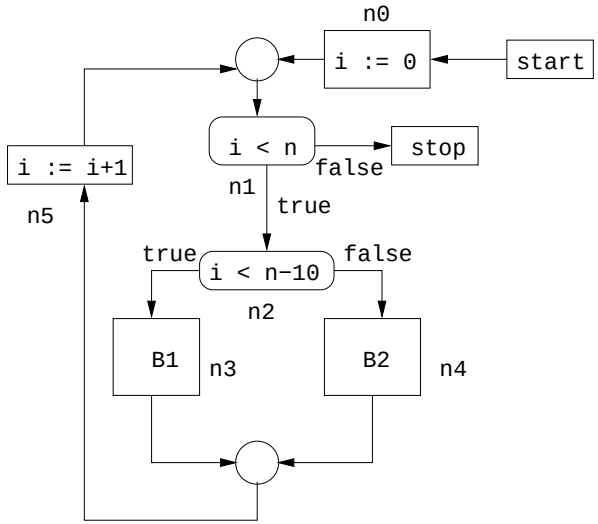
$$
\begin{aligned}
x_i &\geq 0, \quad i = 0, \ldots, 5 \\
x_0 &= 1 \\
x_1 &= x_0 + x_5 \\
x_2 &= x_1 - 1 \\
x_2 &= x_3 + x_4 \\
x_5 &= x_3 + x_4 \\
x_1 &\leq 21
\end{aligned}
$$



Solution $x_0 = 1$, $x_1 = 21$, $x_2 = 20$, $x_3 = 0$, $x_4 = 20$, $x_5 = 20$, $t = 2415$

Second conditional gives new constraints $x_3 \leq 10$, $x_4 \leq 10$:

$$
\begin{aligned}
x_i &\geq 0, \quad i = 0, \ldots, 5 \\
x_0 &= 1 \\
x_1 &= x_0 + x_5 \\
x_2 &= x_1 - 1 \\
x_2 &= x_3 + x_4 \\
x_5 &= x_3 + x_4 \\
x_1 &\leq 21 \\
x_3 &\leq 10 \\
x_4 &\leq 10
\end{aligned}
$$



New solution $x_0 = 1$, $x_1 = 21$, $x_2 = 20$, $x_3 = 10$, $x_4 = 10$, $x_5 = 20$, $t = 1915$

Low-Level Analysis

Use hardware timing model to obtain safe timing estimates for nodes in CFG

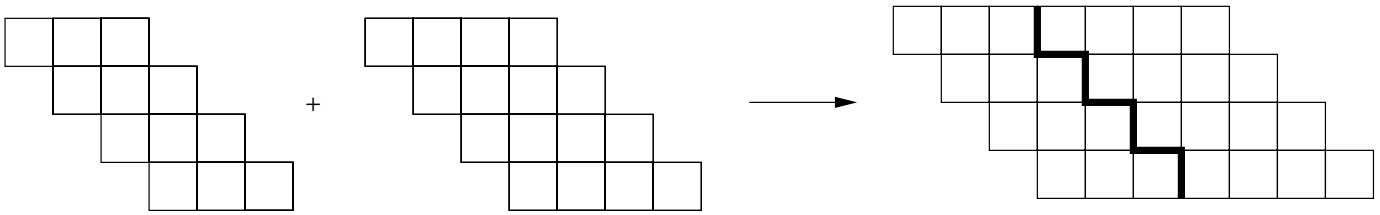
Modern hardware can have complex timing models:

- Pipelined instruction execution
- Caches
- Out-of order instruction execution (superscalar)
- Branch prediction
- etc. . .

Can cause non-additive timing effects (execution time dependent on complex system state)

Caches and *pipelines* are most important to deal with

Pipelined execution starts execution of new instructions before old ones are completed:



Can give overlap effects between execution of program parts:

$T(c_1; c_2) < T(c_1) + T(c2)$

Local effect, can be estimated from execution model for individual instructions

Caches are fast local memories mirroring parts of the main memory

A replacement strategy determines the contents of the cache from the history of memory accesses

Memory access time highly dependent on whether a cache hit or a miss

Cache effects are *global* – a memory access can affect the time of other accesses in completely different parts of the program

A cache analysis estimates, for each program point, the possible cache states

State-of-the art methods use *abstract interpretation* for this

A WCET analysis can then assume worst possible cache state for each access

Flow Analysis

Purpose: to automatically discover constraints on execution counters

Most important to find bounds on execution counters in loops

Early methods were syntax-based: match against library of typical loop patterns, with known iteration count bounds

Current *research* focuses on abstract interpretation as a means to bound values of execution counters

Idea: for each execution counter, find an interval enclosing all its possible values

Current *practice* not so developed: typically, most flow constraints must be provided by hand

Abstract Interpretation

A framework for program analysis (late 70:s, Cousot & Cousot)

It is a *theory of systematic approximations* to obtain computable program analyses

Idea: use elements in some *abstract domain* to represent certain program properties

For instance, a pair of numbers (a, b) to represent that a certain value is restricted to the interval $[a, b]$

The abstract domain must be a complete lattice, with monotone operations

Standard fixed-point theory applies

Program Analysis and Approximations

Program analysis typically aims at deciding a set of possible behaviours, or configurations, for some given program

To find the *exact* set $c(p)$, for any program p, is typically undecidable

May have a method that yields an *approximation* $c'(p)$, where $c(p) \subseteq c'(p)$

Say we want to know that the program never exhibits some *forbidden behaviour* from some set B

If $c'(p) \cap B = \emptyset$, then surely $c(p) \cap B = \emptyset$

Can thus detect cases when surely a forbidden behaviour will not appear

But we may get “false alarms”. Due to undecidability, we have to live with this

Example: Array Bounds Analysis

Consider an array reference `a(i)`

Say a has range $0, \ldots, n$

We want to know for sure that the access never gets out of bounds (a safety property)

Maybe we can find an interval (a, b) containing all possible values of `i`

If $0 \leq a$ and $b \leq n$, then surely the access will never be out of bounds.

For an Ada program (with array bounds checking), this enables a *program optimization*: remove the check

For a C program (without array bounds checking), this is a *formal verification* that the access will never get out of bounds

Representing Properties in Complete Lattices

Complete lattice $(A, \sqsubseteq, \sqcap, \sqcup, \bot, \top)$

Let each element in A represent a property (or set). Then,

- $\sqsubseteq$ represents $\Longrightarrow$, or $\subseteq$ (weakening of information)
- $\sqcap$ approximates $\wedge$ (or $\cap$), and $\sqcup$ approximates $\vee$ (or $\cup$)
- $\bot$ corresponds to contradictory property (alway false), or $\emptyset$
- $\top$ corresponds to trivial property (always true), or universal set

Example: the Complete Lattice of Integer Intervals

Represents certain properties over the integers (or sets of integers)

$A = (Z \cup \{-\infty, \infty\})^2 \quad (= Int)$

$(a, b) \sqsubseteq (a', b') \iff a \geq a' \wedge b \leq b'$

$(a, b) \sqcup (a', b') = (\min(a, a'), \max(b, b'))$

$(a, b) \sqcap (a', b') = (\max(a, a'), \min(b, b'))$

$\bot =$ any (a, b) where $b < a$

$\top = (-\infty, \infty)$

(a, b) represents $p(n) = a \leq n \leq b$

Solutions to Fixed-Point Equations

$f\colon A \to B$ is *monotone* if $a \sqsubseteq a' \implies f(a) \sqsubseteq f(a')$ for all $a, a' \in A$

Let $f\colon A \to A$. Consider the equation $x = f(x)$. This kind of *fixed-point equation* is common in program analysis

If f is monotone, then *Tarski's fixed-point theorem* assures that the equation has a *least fixed-point solution* $\mathrm{lfp}(f)$: for any other solution s, $\mathrm{lfp}(f) \sqsubseteq s$

Fixed-point equations can sometimes be solved by *fixed-point iteration*:

- $f^0 = \bot$
- $f^n = f(f^{n-1})$, $n > 0$

If $f^n = f^{n-1}$ for some n, then $f^n = \mathrm{lfp}(f)$

If fixed-point iteration always terminates, then it can be used as a method to solve fixed-point equations

Termination is guaranteed if the lattice satisfies the *ascending chain condition*: every ascending chain w.r.t. $\sqsubseteq$ must eventually stabilize

Important special case: when A is finite

If the lattice does *not* satisfy the ascending chain condition, then fixed-point iteration may still be possible by an approximation technique called *widening*

This may, however, yield a solution which is not the least fixed-point solution

Concrete and Abstract Lattices

Common situation: a *concrete* lattice, describing exactly some properties of interest, and an *abstract* lattice, with approximate representations of these

Idea: choose abstract lattice such that fixed-point equations always can be solved by fixed-point iteration

Then fixed-point equations, describing properties of interest, can be solved in the abstract lattice rather than in the concrete lattice

Commonly, the concrete lattice is a lattice of sets: $(\mathcal{P}(S), \subseteq, \cap, \cup, \emptyset, S)$

Each $S' \in \mathcal{P}(S)$ then corresponds exactly to a property over S

But if S is infinite, then there are infinite ascending chains w.r.t. $\subseteq$

We then need a well-chosen abstract lattice to solve equations!

Example: Sets of Integers vs. Intervals

Consider $(\mathcal{P}(Z), \subseteq, \cap, \cup, \emptyset, S)$

Describes properties over integers exactly, but cannot be used for analysis

The interval lattice is better for that purpose

How relate integer sets to intervals, and vice versa?

A function $\alpha: \mathcal{P}(Z) \to Int$: $\alpha(S) = (\inf(S), \sup(S))$

α maps S to the *smallest* interval enclosing it

A function $\gamma: Int \to \mathcal{P}(Z)$: $\gamma(a, b) = \{\, z \mid a \leq z \leq b \,\}$

γ maps (a, b) to the *exact* set of integers represented by (a, b)

Galois Connections

Galois connections formalize the relation between concrete and abstract lattice

Let C, A be complete lattices and $\alpha\colon C \to A$, $\gamma\colon A \to C$ monotone functions

α called *abstraction function*, γ *concretization function*

(C, α, γ, A) is then a Galois connection iff:

- $c \sqsubseteq \gamma(\alpha(c))$, $c \in C$
- $a \sqsubseteq \alpha(\gamma(a))$, $a \in A$

Galois connection are at the heart of abstract interpretation!

Some Important Properties of Galois Connections

Transitivity. Makes possible stepwise construction of Galois connections

Cartesian products of lattices with Galois connections has Galois connection

If Galois connection between C and A, then Galois connection between $X \to C$ and $X \to A$, for any set X (total function spaces)

If Galois connections C_1, A_1 and C_2, A_2, then Galois connection between $C_1 \to C_2$ and $A_1 \to A_2$ (monotone function spaces)

An Example

Consider the monotone function space $(\mathcal{P}(Z), \mathcal{P}(Z)) \to \mathcal{P}(Z)$

In particular, addition of sets belongs to this space:

$S_1 + S_2 = \{\, z_1 + z_2 \mid z_1 \in S_1, z_2 \in S_2 \,\}$

There is a Galois connection to the monotone function space from pairs of intervals to intervals. The abstract version of "$+$" on intervals is given by:

$(a_1, b_1) + (a_2, b_2) = (a_1 + a_2, b_1 + b_2)$

Note that $\alpha(S_1 + S_2) \sqsubseteq \alpha(S_1) + \alpha(S_2)$

Operations on concrete lattices can be systematically "lifted" to abstract lattices in this way

Induced Operations, and a Theorem about Fixed-Points

Let $f : C \to C$ be monotone. $\alpha \circ f \circ \gamma : A \to A$ is then the function on A *induced by* f

Theorem If $\alpha \circ f \circ \gamma \sqsubseteq g$, then $\mathrm{lfp}(f) \sqsubseteq \gamma(\mathrm{lfp}(g))$.

Thus, to solve $c = f(c)$ in the concrete domain, we can obtain an upper approximation by instead solving $a = g(a)$ in the abstract domain!

This result is a cornerstone of abstract interpretation

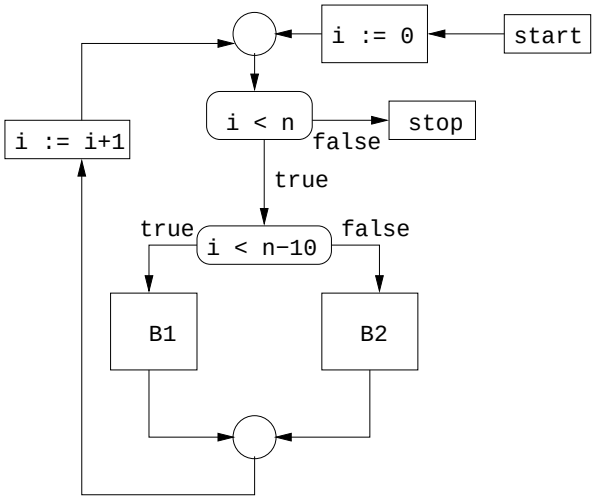
Abstract Interpretation for Imperative Programs

The classical model of Cousot & Cousot (first example of abstract interpretation). Still useful

Imperative programs modeled by flow graphs (essentially same as control flow graphs used by compilers)

Nodes: start, exit, assignment, test, junction

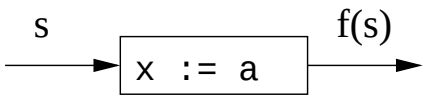
Edges $\sim$ program points



Standard Semantics for Flow Graphs

Program states: mappings from variables to values ($\Sigma = X \to V$)

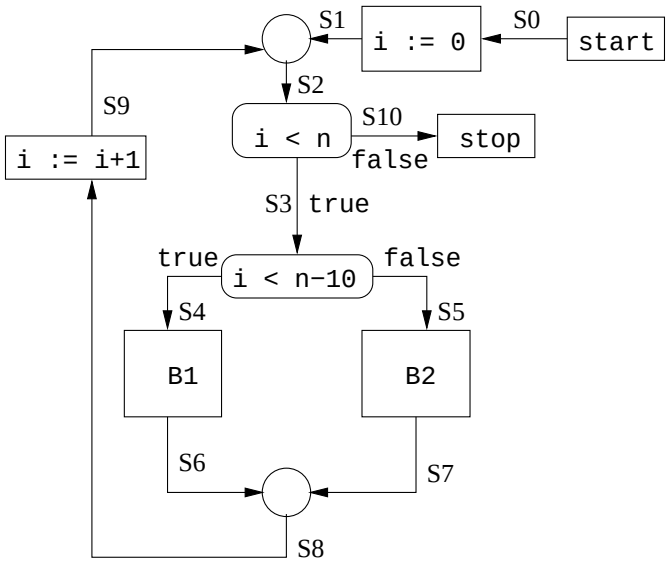
Each node in the flow graph has a transfer function $\Sigma \to \Sigma$ (test node has two)



A standard semantics for a flow graph is then a partial function $\Sigma \to \Sigma$, which for every possible starting state yields its exit state (if terminating)

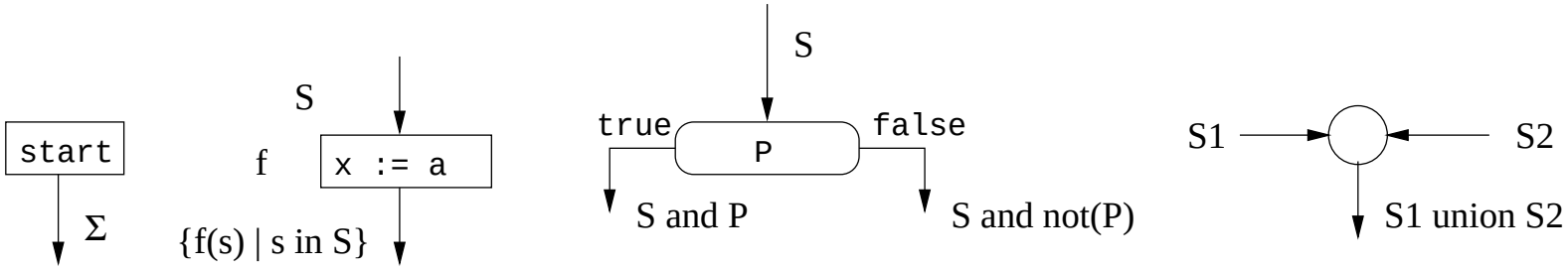
Collecting Semantics for Flow Graphs

Yields, for each program point, the *set* of possible states given that we start in any possible state



Appropriate for program analysis since many interesting properties relate to possible program states

For each type of node, transfer function(s) relating set(s) in to set(s) out:



For each edge out, an equation $S_{out} = F(\vec{S}_{in})$

Yields a system of equations $\vec{S} = \vec{F}(\vec{S})$, where $\vec{S} \in \mathcal{P}(\Sigma)^n$

This is a fixed-point equation over the complete lattice $\mathcal{P}(\Sigma)^n$

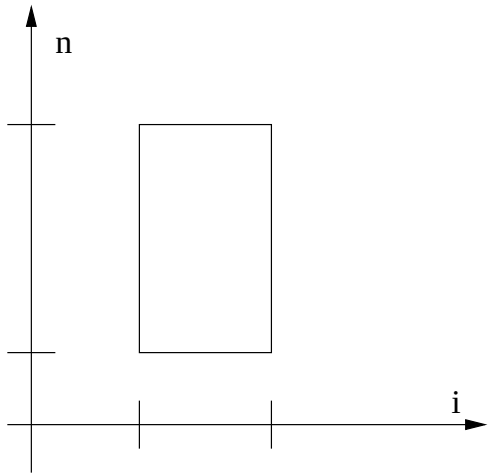
Alas, fixed-point iteration will not terminate in general

Find a Galois connection to an abstract lattice where fixed-point iteration always terminates!

An example: Abstract Interval States

Choose $X \to Int$ as abstract lattice

That is: abstract states, where each variable is assigned an interval rather than a single integer



Abstract interval states represent “rectangle-like” sets of states

There is a Galois connection between $\mathcal{P}(\Sigma)$ and $X \to Int$

The transfer functions for the collecting semantics (sets of states) induce transfer functions for abstract interval states

We obtain $\vec{A} = \vec{G}(\vec{A})$, where $\vec{G}$ is induced from $\vec{F}$

This equation can be solved by fixed-point iteration! (Widening required)

This yields a program analysis, where in each program point the set of possible values for a variable is enclosed in an interval (“interval analysis”)

Can be used for, e.g., array bounds checking

Abstract Interpretation in WCET Flow Analysis

Task: find upper bounds of execution counters for nodes in the flow graph

Can be done with interval analysis

Idea: introduce a virtual *execution counter variable* for each node:

- initialized to zero
- incremented each time the node executes

Find an enclosing interval for each counter variable by interval analysis

Size of interval $=$ upper bound for execution counter

Abstract Interpretation in Low-Level Analysis

Low-level code can also be analyzed by abstract interpretation

The states are more complex: includes contents of registers, memories, pipelines, branch predictors, . . .

The collecting interpretation will concern sets of such states

Again, well-chosen abstract lattices can form the basis for program analyses that find constraints on the possible machine states in different program points

Example: cache (important for performance). An analysis might want to classify a memory access into *sure hit*, *sure miss*, and *don't know*. This can be done by abstract interpretation of the machine states