Lesson 4

Verifying Concurrent Programs
Advanced Critical Section Solutions

Ch 4.1-3, App B [BenA 06]
Ch 5 (no proofs) [BenA 06]

Propositional Calculus
Invariants
Temporal Logic
Automatic Verification
Bakery Algorithm & Variants

Propositional Calculus

(App B [BenA 06])

propositiolaskenta, propositiologiikka totuusarvoilla laskeminen

- Atomic propositions — atominen propositio, tilapropositio
  - A, B, C, …
  - True (T) or False (F)
- Operators
  - not — ei
  - disjunction, or — disjunktio, tai
  - conjunction, and — konjuktio, ja
  - implication — implikaatio
  - equivalence — ekvivalenssi

Boolean algebra

A	$v(A_1)$	$v(A_2)$	$v(A)$
$\neg A_1$	T		F
$\neg A_1$	F		T
$A_1 \vee A_2$	F	F	F
$A_1 \vee A_2$	otherwise		T
$A_1 \wedge A_2$	T	T	T
$A_1 \wedge A_2$	otherwise		F
$A_1 \rightarrow A_2$	T	F	F
$A_1 \rightarrow A_2$	otherwise		T
$A_1 \leftrightarrow A_2$	$v(A_1) = v(A_2)$		T
$A_1 \leftrightarrow A_2$	$v(A_1) \neq v(A_2)$		F

Propositional Calculus

- Implication $(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \rightarrow B$, $A \rightarrow B$ — implikaatio
  - Premise or antecedent — premissit, oletukset
  - Conclusion or consequent — johtopäätös
- Formula — lauseke, argumentti
  - Atomic proposition
  - Atomic propositions or formulaes combined with operators
- Assignment v(f) of formula f — (totuusarvo-) asetus
  - Assigned values (T or F) for each atomic proposition in formula
  - Interpretation v(f) of formula f computed with operator rules
  - Formula f is ***true*** if v(f) = T, ***false*** if v(f)=F

Propositional Calculus

propositiolaskenta

- Formula $(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \rightarrow B$
  - Implication
    - Premise or antecedent — premissit, oletukset
    - Conclusion or consequent — johtopäätös
  - Formula f is true/false if it's interpretation v(f) is true/false — tosi/epätosi
    - Given assignment values for each argument
  - Formula is *valid* if it is *tautology* — pätevä, validi
    - Always true for all interpretations (all atomic propos. values)
  - Formula is *satisfiable* if true in some interpretation — toteutuva
  - Formula is *falsiable* if sometimes false — ei pätevä
  - Formula is *unsatisfiable* if always false — ei toteutuva

Methods for Proving Formulaes Valid

- Induction proof F(n) for all n=1, 2, 3, … — induktio
  - F(1)
  - F(n) → F(n+1)
- Dual approach: f is valid ↔ ¬f is unsatisfiable
  - Find one interpretation that makes ¬f true
    - Go through (automatically) all interpretations of ¬f
    - If such interpretation found, ¬f is satisfiable, i.e., f is not valid — come up with counter example — vasta-esimerkki
    - O/w f is valid
- Proof by contradiction — ristiriita
  - Assume: f is not valid
  - Deduce contradiction with propositional calculus — ¬X ∧ X

Methods for Proving Formulaes Valid

- Deductive proof — deduktiivinen todistus
  - Deduce formula from axioms and existing valid formulaes
  - Start from the "beginning"
- Material implication — "implikaatiotodistus"?
  - Formula is in the form "*p → q*"
  - Can show that "¬(*p → q*)" can not be (or can not become): *v(p)=T* and *v(q)=F*
    - if v(p) = v(q) = T and then if v(q) becomes F, then v(p) will not stay T
    - if v(p) = v(q) = F and then if v(p) becomes T, then v(q) will not stay F

Correctness of Programs

- Program P is <u>partially correct</u>
  - If P halts, then it gives the correct answer
- Program P is <u>totally correct</u>
  - P halts <u>and</u> it gives the correct answer
  - Often <u>very difficult</u> to prove ("halting problem" is difficult)
- Program P can have
  - preconditions A(x1, x2, …) for input values (x1, x2, …)
  - postconditions B(y1, y2, …) for output values (y1, y2, …)
- Partial and total correctness with respect to A(…) and B(…)

More? Se courses on specification and verification

Verification of Concurrent Programs

- State diagrams can be very large
  - Can do them automatically
- Making conclusions on state diagrams is difficult
  - Mutex, no deadlock, no starvation?
  - Can do automatically with temporal logic based on propositional calculus
    - Model checker programs (not covered in this course!)

mallin tarkastin

Spin STeP

Atomic propositions

- Boolean variables (wantp, flag)
  - Consider them as atomic propositions
  - *<u>Proposition</u> wantp* is true, iff *<u>variable</u> wantp* is true in given state
- Integer variables (turn, x)
  - <u>Comparison result</u> is an atomic proposition
  - Example: proposition *"turn ≠ 2"* is true, iff *variable turn* value is not 2 in given state
- Control pointers (p1, p4, q2)
  - <u>Comparison to given value</u> is an atomic proposition
  - Example: proposition *p1* is true, iff *control pointer for P* is p1 in given state

Idea: system state described with propositional logic

Formulaes

Algorithm 3.8: Third attempt

boolean wantp ← false, wantq ← false

	p		q
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	wantp ← true	q2:	wantq ← true
p3:	await wantq = false	q3:	await wantp = false
p4:	critical section	q4:	critical section
p5:	wantp ← false	q5:	wantq ← false

- Formula: p1 ∧ q1 ∧ ¬wantp ∧ ¬wantq
  - True only in the starting state
- Formula: p4 ∧ q4
  - True only if mutex is broken
  - Mutex condition can be <u>defined</u>: ¬(p4 ∧ q4)
    - Must be true in all possible states in all possible computations
    - <u>Invariant</u>

invariantti

Mutex Proof

Algorithm 3.8: Third attempt

boolean wantp ← false, wantq ← false

	p		q
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	wantp ← true	q2:	wantq ← true
p3:	await wantq = false	q3:	await wantp = false
p4:	critical section	q4:	critical section
p5:	wantp ← false	q5:	wantq ← false

invariantti, aina tosi

- Invariant ¬(p4 ∧ q4)
  - If this is proven correct (true in all states), then mutex is proven
- Inductive proof
  - True for *initial state*
  - Assuming true for *current state*, prove that it still applies in *next state*
    - Consider <u>only statements</u> that affect <u>propositions in invariant</u>

Mutex Proof

Algorithm 3.8: Third attempt

boolean wantp ← false, wantq ← false

	p		q
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	wantp ← true	q2:	wantq ← true
p3:	await wantq = false	q3:	await wantp = false
p4:	critical section	q4:	critical section
p5:	wantp ← false	q5:	wantq ← false

- Invariant *¬(p4 ∧ q4)*
  - Can not prove directly (yet) – too difficult
- Need proven Lemma 4.3
  - Lemma 4.1: *p3..5 → wantp* is invariant
  - Lemma 4.2: *wantp → p3..5* is invariant
  - Lemma 4.3: *p3..5 ↔ wantp* and q*3..5 ↔ wantq* are invariants
  - Proof not covered here
- Can now prove original invariant *¬(p4 ∧ q4)*
  - Inductive proof with Lemma 4.3
  - Details on next slide

lemma, apulause

Mutex Proof

Algorithm 3.8: Third attempt

boolean wantp ← false, wantq ← false

p		q	
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	wantp ← true	q2:	wantq ← true
p3:	await wantq = false	q3:	await wantp = false
p4:	critical section	q4:	critical section
p5:	wantp ← false	q5:	wantq ← false

- **Lemma 4.3**: *p3..5 ↔ wantp* and q*3..5 ↔ wantq* invariants
- **Theorem 4.4**: *¬(p4 ∧ q4)* is invariant
  - Prove *(p4 ∧ q4)* inductively false in every state
  - Initial state: trivial
  - Only states {p3, …} need to be considered
    - p4 may become true only here, i.e., state {p4, q?, …}
    - States {…, q3, …} similar, symmetrical
  - Can execute {p3, …} only if wantq=false (i.e., ¬wantq)
    - Because wantq=false, q4 is also false (Lemma 4.3)
    - Next state can not be {p4, q4, …}, i.e., *(p4 ∧ q4)* is false ∎

24.1.2011 13

Temporal Logic

temporaalilogiikka, aikaperustainen logiikka

- Propositional logic with extra temporal operators
- Computation $\{s_0, s_1, s_2, \ldots\}$
  - Infinite sequence of states: $\{s_0, s_1, s_2, \ldots\}$
- Temporal operators
  - Value (T or F) of given predicate does not necessarily depend only on current state
    - It may depend on also on (some or all) future states
  - Always or box (□) operator — aina
    - □A true in state s_i if A true in all s_j, $j \geq i$ — □¬(p4 ∧ q4)
    - E.g., mutex must always be true
  - Eventually or diamond (◊) operator — lopulta, joskus tulevaisuudessa
    - ◊A true in state s_i if A true in some s_j, $j \geq i$ — □(p2 → ◊p4)
    - E.g., no starvation means that something eventually will become true

24.1.2011 14

Other Temporal Logic Operators

seuraavassa tilassa

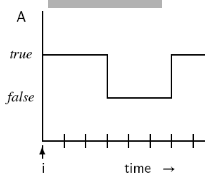
- True in next state (O) operator
  - Op true in state s_i, if p is true in the state s_{i+1}
- Until eventually (**U**) operator — tosi kunnes, kunnes lopulta
  - p **U** q true in state s_i, if p is true in every state in future until eventually q becomes true
- …
- Not used (needed) in this course…

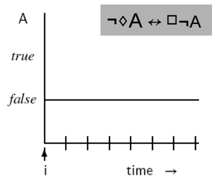
More? See courses on specification and verification.

24.1.2011 15

Some Laws of Temporal Logic

- deMorgan — ¬(A ∧ B) ↔ (¬A ∨ ¬B) — ¬(A ∨ B) ↔ (¬A ∧ ¬B)
- Distributive Laws — vaihdantalaki — □(A ∧ B) ↔ (□A ∧ □B) — ◊(A ∨ B) ↔ (◊A ∨ ◊B)
- Duality — dualiteetti
  - Not always is equivalent to eventually not — ¬□A ↔ ◊¬A
  - Not eventually is equivalent to always not — ¬◊A ↔ □¬A

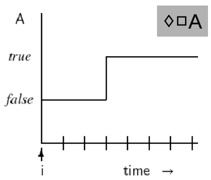


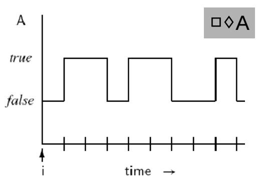


24.1.2011 16

Sequence

- Eventually always ◊□A — lopulta aina, joskus tulevaisuudessa pysyvästi totta
  - Will come true and then stays true forever
- Always eventually □◊A — aina lopulta, äärettömän usein tulevaisuudessa
  - Always will become true some times in future (again)





24.1.2011 17

More Complex Proofs

- State diagrams become easily too large for manual analysis
- Use model checkers
  - Spin for Promela programs (algorithms)
  - Java PathFinder for Java programs
- More details?
  - Course *An Introduction to Specification and Verification*

Spesifioinnin ja verifioinnin perusteet

24.1.2011 18

Advanced Critical Section Solutions

Ch 5 [BenA 06] (no proofs)

Bakery Algorithm

Bakery for N processes

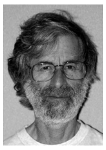
Fast for N processes

Bakery Algorithm

(Leslie Lamport)

numerolappualgoritmi

- Environment
  - Shared memory, atomic read/write
    - No HW support needed — Very strong requirement!
  - Short exclusive access code segments
    - Wait in busy loop (no process switch)
- Goal
  - Mutex *and* Customers served in request order
  - Independent (distributed) decision making
- Solution idea
  - Get queue number, service requests in ascending order
- Possible problems
  - Shared, distributed queuing machine, will it work?
  - Get same queue number as someone else? Problem?
  - Some number skipped? Problem or not?
  - Will numbers grow indefinitely (overflow)?



Bakery Algorithm (2 processes)

Algorithm 5.1: Bakery algorithm (two processes)

integer np ← 0, nq ← 0

p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: np ← nq + 1	q2: nq ← np + 1
p3: await nq = 0 or np ≤ nq	q3: await np = 0 or nq < np
p4: critical section	q4: critical section
p5: np ← 0	q5: nq ← 0

In real life usually not atomic!

q in non-critical section

q in q3 or q4

- Can enter CS, if ticket (np or nq) is "smaller" than that of the other process
- Priority: if equal tickets, both compete, but P wins
  - Fixed priority not so good, but acceptable (rare occurrence)

Discuss

Correctness Proof for 2-process Bakery Algorithm

Alg. 5.1

- Mutex?
- No deadlock?
- No starvation?
- No counter overflow?

- What else, if any?

- How?
  - Temporal logic

Spesifioinnin ja verifioinnin perusteet

(Slides Conc.Progr. 2006)

(for those who really like temporal logic…)

Bakery for n Processes

Algorithm 5.2: Bakery algorithm (*N* processes)

integer array[1..n] number ← [0,...,0]

```
    loop forever
p1:     non-critical section
p2:     number[i] ← 1 + max(number)
p3:     for all other processes j
p4:         await (number[j] = 0) or (number[i] ≪ number[j])
p5:     critical section
p6:     number[i] ← 0
```

not atomic!?

when equality, give priority to smaller number[x]

in non-critical section?

in q3..q6?

- No write competition to shared variables
  - Load/store assumed atomic
- Ticket numbers increase continuously while critical section is taken – danger?
- All other processes polled
  - Not so good!

Bakery for n Processes

Alg. 5.2

- Mutex OK?
  - Yes, because of priorities at competition time
- Deadlock OK?
  - Yes, because of priorities at competition time
- Starvation OK?
  - Yes, because
    - Your (i) turn will come eventually
    - Others (j) will progress and leave CS
    - Next time their number[j] will be bigger than yours
- Overflow
  - Not good. Numbers grow unbounded if some process always in CS
    - Must have other information/methods to guarantee that this does not happen.

e.g., max 100 processes, CS less than 0.01% of executed code ??

Algorithm 5.3: Bakery algorithm without atomic assignment (3)

boolean array[1..n] choosing ← [false,. . . ,false]
integer array[1..n] number ← [0,. . . ,0]

```
     loop forever
p1:     non-critical section
p2:     choosing[i] ← true
p3:     number[i] ← 1 + max(number)
p4:     choosing[i] ← false
p5:     for all other processes j
p6:        await choosing[j] = false
p7:        await (number[j] = 0) or (number[i] ≪ number[j])
p8:     critical section
p9:     number[i] ← 0
```

critical section within entry protocol to critical section…

do not read number[j] when j is changing it

what if j is real fast: p9, p1,.., p3 ?

- Concurrent read & write may result to bad read
- Lamport, 1974
  - Correct behaviour in p7 even if number[j] value read wrong!
    - Assuming that await is in busy loop

http://research.microsoft.com/users/lamport/pubs/bakery.pdf click

Performance Problems with Bakery Algorithm

- Problem
  - Lots of overhead work, if many concurrent processes
  - Check status for all possibly competing other processes
    - Other processes (not in CS) slow down the one process trying to get into CS – not good
  - Most of the time wasted work
    - Usually not much competition for CS
- How to do it better?
  - Check competition in fixed time
  - In a way not dependent on the number of possible competitors
  - Suffer overhead only when competition occurs

Algorithm 5.4: Fast algorithm for two processes (outline)

integer gate1 ← 0, gate2 ← 0

p		q	
	loop forever		loop forever
	non-critical section		non-critical section
p1:	gate1 ← p	q1:	gate1 ← q
p2:	if gate2 ≠ 0 goto p1	q2:	if gate2 ≠ 0 goto q1
p3:	gate2 ← p	q3:	gate2 ← q
p4:	if gate1 ≠ p	q4:	if gate1 ≠ q
p5:	if gate2 ≠ p goto p1	q5:	if gate2 ≠ q goto q1
	critical section		critical section
p6:	gate2 ← 0	q6:	gate2 ← 0

- Assume atomic read/write
- 2 shared variables, both read/written by P and Q
- Block at gate1, if contention
  - Last one to get there waits
- Access to CS, if success in writing own id to both gates

Algorithm 5.4: Fast algorithm for two processes (outline)

integer gate1 ← 0, gate2 ← 0

p		q	
	loop forever		loop forever
	non-critical section		non-critical section
p1:	gate1 ← p	q1:	gate1 ← q
p2:	if gate2 ≠ 0 goto p1	q2:	if gate2 ≠ 0 goto q1
p3:	gate2 ← p	q3:	gate2 ← q
p4:	if gate1 ≠ p	q4:	if gate1 ≠ q
p5:	if gate2 ≠ p goto p1	q5:	if gate2 ≠ q goto q1
	critical section		critical section
p6:	gate2 ← 0	q6:	gate2 ← 0

- No contention for P, if P alone (i.e., gate2 =0)
  - Little overhead in entry
    - 2 assignments and 2 comparisons

Algorithm 5.4: Fast algorithm for two processes (outline)

integer gate1 ← 0, gate2 ← 0

p		q	
	loop forever		loop forever
	non-critical section		non-critical section
p1:	gate1 ← p	q1:	gate1 ← q
p2:	if gate2 ≠ 0 goto p1	q2:	if gate2 ≠ 0 goto q1
p3:	gate2 ← p	q3:	gate2 ← q
p4:	if gate1 ≠ p	q4:	if gate1 ≠ q
p5:	if gate2 ≠ p goto p1	q5:	if gate2 ≠ q goto q1
	critical section		critical section
p6:	gate2 ← 0	q6:	gate2 ← 0

- Q pass gate2 (q3), when P tries to get in
  - P blocks at p2, until Q releases gate2
  - Q will advance even if P gets to p1 before q4 executed

Algorithm 5.4: Fast algorithm for two processes (outline) (2)

```
integer gate1 ← 0, gate2 ← 0
            p                                   q
    loop forever                        loop forever
       non-critical section                non-critical section
p1:    gate1 ← p                   q1:     gate1 ← q
p2:    if gate2 ≠ 0 goto p1        q2:     if gate2 ≠ 0 goto q1
p3:    gate2 ← p                   q3:     gate2 ← q
p4:    if gate1 ≠ p                q4:     if gate1 ≠ q
p5:       if gate2 ≠ p goto p1     q5:        if gate2 ≠ q goto q1
       critical section                    critical section
p6:    gate2 ← 0                   q6:     gate2 ← 0
```

gate1 gate2: p, 0 → p, q → ok / ok

- Q arrives at the same time with P
  - Competition on who wrote to gate1 and gate2 <u>last</u>
  - P & P: P advances, Q blocks at q5
  - P & Q; P advances, Q advances, i.e., no mutex (ouch!)

Algorithm 5.6: Fast algorithm for two processes (2)

```
integer gate1 ← 0, gate2 ← 0
boolean wantp ← false, wantq ← false
            p                                   q
p1:    gate1 ← p                   q1:     gate1 ← q
       wantp ← true                        wantq ← true
p2:    if gate2 ≠ 0                q2:     if gate2 ≠ 0
          wantp ← false                       wantq ← false
          goto p1                             goto q1
p3:    gate2 ← p                   q3:     gate2 ← q
p4:    if gate1 ≠ p                q4:     if gate1 ≠ q
          wantp ← false                       wantq ← false
          await wantq = false                 await wantp = false
p5:       if gate2 ≠ p goto p1     q5:        if gate2 ≠ q goto q1
          else wantp ← true                   else wantq ← true
       critical section                    critical section
p6:    gate2 ← 0                   q6:     gate2 ← 0
       wantp ← false                       wantq ← false
```

P last at gate1
Q last at gate 2

Q blocks here

Fast N Process Baker

- Expand Alg. 5.6
  - Still with just 2 gates

Alg. 5.6

P: await wantq=false → Pi: For all other j await want[j]=false

- Still fast, even with "for all other"
  - Fast when no contention (gate2 = 0)
    - Entry: 3 assignments, 2 if's
  - Awaits done only when contention
    - p4: if gate1 ≠ i

Summary

- How to verify concurrent programs with Propositional Calculus and Temporal Logic
- Use of invariants in correctness proofs
  - E.g., mutual exclusion (mutex) proofs with invariants
  - Can often use in practice, when no formal proofs used
- Bakery algorithm
  - Shared memory
  - No HW support for concurrency control
  - 2 or N processes
  - Overflow problem, performance problem