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

- Atomic propositions
  - A, B, C, ...
  - True (T) or False (F)
- Operators
  - not
  - disjunction, or
  - conjunction, and
  - implication
  - equivalence

Boolean algebra

<table>
<thead>
<tr>
<th>A</th>
<th>v(A₁)</th>
<th>v(A₂)</th>
<th>v(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬A₁</td>
<td>T</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>¬A₁</td>
<td>F</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>A₁ ∨ A₂</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>A₁ ∨ A₂</td>
<td>F</td>
<td>otherwise</td>
<td>T</td>
</tr>
<tr>
<td>A₁ ∧ A₂</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>A₁ ∧ A₂</td>
<td>T</td>
<td>otherwise</td>
<td>F</td>
</tr>
<tr>
<td>A₁ → A₂</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>A₁ → A₂</td>
<td>T</td>
<td>otherwise</td>
<td>T</td>
</tr>
<tr>
<td>A₁ ↔ A₂</td>
<td>v(A₁) = v(A₂)</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>A₁ ↔ A₂</td>
<td>v(A₁) ≠ v(A₂)</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

(App B [BenA 06])

propositiolaskenta, propositiologiikka
totuusarvoilla laskeminen

atominen propositio, tilapropositio

24.1.2011 Copyright Teemu Kerola 2011
Propositional Calculus

- **Implication**
  \[(A_1 \land A_2 \land \cdots \land A_n) \rightarrow B\]

  - Premise or antecedent
  - Conclusion or consequent
  - Formula

- **Assignment** \(v(f)\) of formula \(f\)
  - 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

- **Formula**
  \[(A_1 \land A_2 \land \cdots \land A_n) \rightarrow B\]

  - Implication
    - Premise or antecedent
    - Conclusion or consequent
  - Formula is **valid** if it is tautology
    - Always true for all interpretations (all atomic propos. values)
  - Formula is **satisfiable** if true in some interpretation
  - Formula is **falseable** if sometimes false
  - Formula is **unsatisfiable** if always false
Methods for Proving Formulae Valid

- Induction proof \( F(n) \) for all \( n=1, 2, 3, ... \)
  - \( F(1) \)
  - \( F(n) \to F(n+1) \)
- Dual approach: \( f \) is valid \( \iff \) \( \neg f \) is unsatisfiable
  - Find one interpretation that makes \( \neg f \) true
  - Go through (automatically) all interpretations of \( \neg f \)
  - If such interpretation found, \( \neg f \) is satisfiable, i.e., \( f \) is not valid
  - Otherwise \( f \) is valid
- Proof by contradiction
  - Assume: \( f \) is not valid
  - Deduce contradiction with propositional calculus

\[ \neg X \land X \]

Methods for Proving Formulae Valid

- Deductive proof
  - Deduce formula from axioms and existing valid formulae
  - Start from the “beginning”

- Material implication
  - Formula is in the form “\( p \to q \)”
  - Can show that “\( \neg(p \to q) \)” 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 partially correct**
  - If P halts, then it gives the correct answer
- **Program P is totally correct**
  - P halts and it gives the correct answer
  - Often very difficult 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(...)
Atomic propositions

- **Boolean variables**
  - Consider them as atomic propositions
  - Proposition `wantp` is true, iff variable `wantp` is true in given state
- **Integer variables**
  - Comparison result is an atomic proposition
  - Example: proposition “turn ≠ 2” is true, iff variable `turn` value is not 2 in given state
- **Control pointers**
  - Comparison to given value 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

Formulae

**Algorithm 3.8: Third attempt**

<table>
<thead>
<tr>
<th>boolean wantp ← false, wantq ← false</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
</tr>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
</tr>
<tr>
<td>p2: wantp ← true</td>
</tr>
<tr>
<td>p3: await wantq = false</td>
</tr>
<tr>
<td>p4: critical section</td>
</tr>
<tr>
<td>p5: wantp ← false</td>
</tr>
</tbody>
</table>

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

24.1.2011 Copyright Teemu Kerola 2011
### Mutex Proof

#### Algorithm 3.8: Third attempt

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p1:</td>
<td>wantp ← true</td>
<td>q1:</td>
</tr>
<tr>
<td>p2:</td>
<td>await wantq = false</td>
<td>q2: wantq ← true</td>
</tr>
<tr>
<td>p3:</td>
<td>critical section</td>
<td>q3: await wantp = false</td>
</tr>
<tr>
<td>p4:</td>
<td>wantp ← false</td>
<td>q4: critical section</td>
</tr>
<tr>
<td>p5:</td>
<td>wantp ← false</td>
<td>q5: wantq ← false</td>
</tr>
</tbody>
</table>

- Invariant \(\neg(p4 \land q4)\)
  - 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 only statements that affect propositions in invariant

#### Mutex Proof

- Invariant \(\neg(p4 \land q4)\)
  - Can not prove directly (yet) – too difficult
- Need proven Lemma 4.3
  - Lemma 4.1: \(p3..5 \rightarrow wantp\) is invariant
  - Lemma 4.2: \(wantp \rightarrow p3..5\) is invariant
  - Lemma 4.3: \(p3..5 \leftrightarrow wantp\) and \(q3..5 \leftrightarrow wantq\) are invariants
  - Proof not covered here
- Can now prove original invariant \(\neg(p4 \land q4)\)
  - Inductive proof with Lemma 4.3
  - Details on next slide
Mutex Proof

Lemma 4.3: \( p_{3..5} \Rightarrow \text{wantp} \) and \( q_{3..5} \Rightarrow \text{wantq} \) invariants

Theorem 4.4: \( \neg(p_{4} \land q_{4}) \) is invariant
- Prove \((p_{4} \land q_{4})\) inductively false in every state
- Initial state: trivial
- Only states \(\{p_{3}, \ldots\}\) need to be considered
  - \(p_{4}\) may become true only here, i.e., state \(\{p_{4}, q_{?}, \ldots\}\)
  - States \(\{\ldots, q_{3}, \ldots\}\) similar, symmetrical
- Can execute \(\{p_{3}, \ldots\}\) only if \(\neg\text{wantq}\)
  - Because \(\text{wantq}=\text{false}\), \(q_{4}\) is also false (Lemma 4.3)
  - Next state can not be \(\{p_{4}, q_{4}, \ldots\}\), i.e., \((p_{4} \land q_{4})\) is false

Temporal Logic

- 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 (\(\square\)) operator
    - \(\square A\) true in state \(s_{i}\) if \(A\) true in all \(s_{j}, j \geq i\)
    - E.g., mutex must always be true
  - Eventually or diamond (\(\diamond\)) operator
    - \(\diamond A\) true in state \(s_{i}\) if \(A\) true in some \(s_{j}, j \geq i\)
    - E.g., no starvation means that something eventually will become true
Other Temporal Logic Operators

- True in next state (O) operator
  - Op true in state si, if p is true in the state si+1

- Until eventually (U) operator
  - p Uq true in state si, 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.

Some Laws of Temporal Logic

- deMorgan
  - ¬(A ∧ B) ↔ (¬A ∨ ¬B)
  - ¬(A ∨ B) ↔ (¬A ∧ ¬B)

- Distributive Laws
  - □(A ∧ B) ↔ (□A ∧ □B)
  - ◇(A ∨ B) ↔ (◇A ∨ ◇B)

- Duality
  - Not always is equivalent to eventually not
  - ¬◇A ↔ ◇¬A
  - Not eventually is equivalent to always not
  - ¬□A ↔ □¬A
Lecture 4: Verifying Solutions and Turn-Ticket Problem

Sequence

- Eventually always \[\Box A\] lopulta aina, joskus tulevaisuudessa pysyvästi totta
  - Will come true and then stays true forever
- Always eventually \[\Diamond A\] aina lopulta, äärettömän usein tulevaisuudessa
  - Always will become true some times in future (again)

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*
Advanced Critical Section Solutions

Ch 5 [BenA 06] (no proofs)

Bakery Algorithm
Bakery for N processes
Fast for N processes
Bakery Algorithm

- Environment
  - Shared memory, atomic read/write
    - No HW support needed
  - 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)

\[
\begin{align*}
\text{loop forever} & \quad \text{loop forever} \\
p_1: & \quad \text{non-critical section} \quad \text{non-critical section} \\
p_2: & \quad np \leftarrow np + 1 \\
p_3: & \quad \text{await } nq = 0 \text{ or } np \preceq nq \\
p_4: & \quad \text{critical section} \\
p_5: & \quad np \leftarrow 0 \\
q_1: & \quad \text{non-critical section} \\
q_2: & \quad nq \leftarrow np + 1 \\
q_3: & \quad \text{await } np = 0 \text{ or } nq \preceq np \\
q_4: & \quad \text{critical section} \\
q_5: & \quad nq \leftarrow 0
\end{align*}
\]

- 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)
Correctness Proof for 2-process Bakery Algorithm

- Mutex?
- No deadlock?
- No starvation?
- No counter overflow?
- What else, if any?

How?
- 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

- 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

- 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.

Alg. 5.2

Algorithm 5.3: Bakery algorithm without atomic assignment (3)

```java
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] ← i + 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
```
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)

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>gate1 ← p</td>
<td>q1: gate1 ← q</td>
</tr>
<tr>
<td>p2</td>
<td>if gate2 ≠ 0 goto p1</td>
<td>q2: if gate2 ≠ 0 goto q1</td>
</tr>
<tr>
<td>p3</td>
<td>gate2 ← p</td>
<td>q3: gate2 ← q</td>
</tr>
<tr>
<td>p4</td>
<td>if gate1 ≠ p</td>
<td>q4: if gate1 ≠ q</td>
</tr>
<tr>
<td>p5</td>
<td>if gate2 ≠ p goto p1</td>
<td>q5: if gate2 ≠ q goto q1</td>
</tr>
<tr>
<td></td>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>p6</td>
<td>gate2 ← 0</td>
<td>q6: gate2 ← 0</td>
</tr>
</tbody>
</table>

- 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)

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td></td>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p1</td>
<td>gate1 ← p</td>
<td>q1: gate1 ← q</td>
</tr>
<tr>
<td>p2</td>
<td>if gate2 ≠ 0 goto p1</td>
<td>q2: if gate2 ≠ 0 goto q1</td>
</tr>
<tr>
<td>p3</td>
<td>gate2 ← p</td>
<td>q3: gate2 ← q</td>
</tr>
<tr>
<td>p4</td>
<td>if gate1 ≠ p</td>
<td>q4: if gate1 ≠ q</td>
</tr>
<tr>
<td>p5</td>
<td>if gate2 ≠ p goto p1</td>
<td>q5: if gate2 ≠ q goto q1</td>
</tr>
<tr>
<td>p6</td>
<td>gate2 ← 0</td>
<td>q6: gate2 ← 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>critical section</td>
</tr>
</tbody>
</table>

- 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)

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td></td>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p1</td>
<td>gate1 ← p</td>
<td>q1: gate1 ← q</td>
</tr>
<tr>
<td>p2</td>
<td>if gate2 ≠ 0 goto p1</td>
<td>q2: if gate2 ≠ 0 goto q1</td>
</tr>
<tr>
<td>p3</td>
<td>gate2 ← p</td>
<td>q3: gate2 ← q</td>
</tr>
<tr>
<td>p4</td>
<td>if gate1 ≠ p</td>
<td>q4: if gate1 ≠ q</td>
</tr>
<tr>
<td>p5</td>
<td>if gate2 ≠ p goto p1</td>
<td>q5: if gate2 ≠ q goto q1</td>
</tr>
<tr>
<td>p6</td>
<td>gate2 ← 0</td>
<td>q6: gate2 ← 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>critical section</td>
</tr>
</tbody>
</table>

- 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
Lecture 4: Verifying Solutions and Turn-Ticket Problem

- Q arrives at the same time with P
  - Competition on who wrote to gate1 and gate2 last
  - P & P: P advances, Q blocks at q5
  - P & Q: P advances, Q advances, i.e., no mutex (ouch!)
Fast N Process Baker

- Expand Alg. 5.6
  - Still with just 2 gates

  P: \text{await wantq=false} \quad \rightarrow \quad \text{Pi: For all other j, 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 \neq 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