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

<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>F</td>
<td>F</td>
</tr>
<tr>
<td>¬A1</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>A1 ∨ A2</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>A1 ∨ A2</td>
<td>otherwise</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>A1 ∧ A2</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>A1 ∧ A2</td>
<td>otherwise</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>A1 ⟹ A2</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>A1 ⟹ A2</td>
<td>otherwise</td>
<td>F</td>
<td></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])

propositional calculus, propositional logic, truth table
atomic proposition, bivalent calculus

Boolean algebra

Lecture 4: Verifying Solutions and Turn-Ticket Problem
Propositional Calculus

- Implication \((A_1 \land A_2 \land \cdots \land A_n) \rightarrow B\)
  - Premise or antecedent
  - Conclusion or consequent

- Formula
  - Atomic proposition
  - Atomic propositions or formulae combined with operators

- 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 \(v(f) = F\)
Methods for Proving Formulae Valid

- Induction proof \( F(n) \) for all \( n=1, 2, 3, ... \)
  - \( F(1) \)
  - \( F(n) \rightarrow F(n+1) \)

- 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
    - \( \lor/w \) \( f \) 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 "beginnin"

- Material implication
  - Formula is in the form \( p \rightarrow q \)
  - Can show that \( \neg(p \rightarrow q) \) can not be (or can not become): \( v(p)=T \) and \( v(q)=F \)
    - if \( v(p)=v(q)=T \) and \( v(q) \) becomes \( F \), then \( v(p) \) will not stay \( T \)
    - if \( v(p)=v(q)=F \) and \( v(p) \) becomes \( T \).
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(....)

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
      - Spin
      - STeP

More? See courses on specification and verification.
Atomic propositions

• Boolean variables
  - Consider them as atomic propositions
  - Proposition \( \text{wantp} \) is true, iff variable \( \text{wantp} \) is true in given state

• Integer variables
  - Comparison result is an atomic proposition
  - Example: proposition "turn \( \neq 2 \)" 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

Formulaes

Algorithm 3.8: Third attempt

| boolean \( \text{wantp} \) ← false, \( \text{wantq} \) ← false |
|-----------------|-----------------|
| \( p \) | \( q \) |
| loop forever | loop forever |
| \( p1 \): non-critical section | \( q1 \): non-critical section |
| \( p2 \): \( \text{wantp} \) ← true | \( q2 \): \( \text{wantq} \) ← true |
| \( p3 \): await \( \text{wantq} \) = false | \( q3 \): await \( \text{wantp} \) = false |
| \( p4 \): \( \text{critical section} \) | \( q4 \): \( \text{critical section} \) |
| \( p5 \): \( \text{wantp} \) ← false | \( q5 \): \( \text{wantq} \) ← false |

• Formula: \( p1 \land q1 \\
  \land \neg \text{wantp} \\
  \land \neg \text{wantq} \)
  - True only in the starting state

• Formula: \( p4 \land q4 \)
  - True only if mutex is broken
  - Mutex condition can be defined: \( \neg (p4 \land q4) \)
    • Must be true in all possible states in all possible computations
    • Invariant

4.11.2008 Copyright Teemu Kerola 2008
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>loop forever</td>
<td></td>
</tr>
<tr>
<td>p1:</td>
<td>non-critical section</td>
<td>q1:</td>
</tr>
<tr>
<td>p2:</td>
<td>wantp ← true</td>
<td>q2:</td>
</tr>
<tr>
<td>p3:</td>
<td>await wantq = false</td>
<td>q3:</td>
</tr>
<tr>
<td>p4:</td>
<td>critical section</td>
<td>q4:</td>
</tr>
<tr>
<td>p5:</td>
<td>wantp ← false</td>
<td>q5:</td>
</tr>
</tbody>
</table>

- 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: \( p3..5 \Rightarrow wantp \) and \( q3..5 \Rightarrow wantq \) are invariants
- Can now prove original invariant \( \neg(p4 \land q4) \)
  - Inductive proof with Lemma 4.3
  - Details on next slide


**Mutex Proof**

**Lemma 4.3:** \( p_{3..5} \leftrightarrow \text{want}_p \) and if \( p_{3..5} \leftrightarrow \text{want}_q \) are 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 \( \text{want}_q = \text{false} \) (i.e., \( \neg\text{want}_q \))
  - Because \( \text{want}_q = \text{false} \), \( q_4 \) 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
  - 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
    - \( [A] \) 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
    - \( \langle A \rangle \) 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
  - $O \, p$ true in state $s_i$, if $p$ is true in the state $s_{i+1}$

- Until eventually ($U$) operator
  - $p \, U \, q$ 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.

Some Laws of Temporal Logic

- deMorgan
  - $\neg(A \land B) \iff (\neg A \lor \neg B)$
  - $\neg(A \lor B) \iff (\neg A \land \neg B)$

- Distributive Laws
  - $\Box(A \land B) \iff (\Box A \land \Box B)$
  - $\Diamond(A \lor B) \iff (\Diamond A \lor \Diamond B)$

- Duality
  - Not always is equivalent to eventually not
    - $\neg \Diamond A \iff \Box \neg A$
  - Not eventually is equivalent to always not
    - $\neg \Box A \iff \Diamond \neg A$
Sequence

- Eventually always \( \diamond \Box \) lopulta aina, joskus tulevaisuudessa pysyvästi totta
  - Will come true and then stays true forever
- Always eventually \( \Box \diamond \) 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*
    Spesifioinnin ja verifioinnin perusteet
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)**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
<td>q1: non-critical section</td>
</tr>
<tr>
<td>p2: $np \leftarrow np + 1$</td>
<td>q2: $nq \leftarrow np + 1$</td>
</tr>
<tr>
<td>p3: await $nq = 0$ or $np &lt; nq$</td>
<td>q3: await $np = 0$ or $nq &lt; np$</td>
</tr>
<tr>
<td>p4: critical section</td>
<td>q4: critical section</td>
</tr>
<tr>
<td>p5: $np \leftarrow 0$</td>
<td>q5: $nq \leftarrow 0$</td>
</tr>
</tbody>
</table>

- 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)

<table>
<thead>
<tr>
<th>Loop forever</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: non-critical section</td>
</tr>
<tr>
<td>p2: number[i] ← 1 + max(number)</td>
</tr>
<tr>
<td>p3: for all other processes j</td>
</tr>
<tr>
<td>p4: await (number[j] = 0) or (number[i] &lt; number[j])</td>
</tr>
<tr>
<td>p5: critical section</td>
</tr>
<tr>
<td>p6: number[i] ← 0</td>
</tr>
</tbody>
</table>

- 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
do not happen.

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

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] ← 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
```

- 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


Lecture 4: Verifying Solutions and Turn-Ticket Problem
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>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p1: gate1 ← p</td>
<td>q1: gate1 ← q</td>
</tr>
<tr>
<td>p2: if gate2 ≠ 0 goto p1</td>
<td>q2: if gate2 ≠ 0 goto q1</td>
</tr>
<tr>
<td>p3: gate2 ← p</td>
<td>q3: gate2 ← q</td>
</tr>
<tr>
<td>p4: if gate1 ≠ p</td>
<td>q4: if gate1 ≠ q</td>
</tr>
<tr>
<td>p5: if gate2 ≠ p goto p1</td>
<td>q5: if gate2 ≠ q goto q1</td>
</tr>
<tr>
<td></td>
<td>critical section</td>
</tr>
<tr>
<td></td>
<td>critical section</td>
</tr>
<tr>
<td>p6: 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>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer gate1 ← 0, gate2 ← 0</td>
<td></td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p1: gate1 ← p</td>
<td>q1: gate1 ← q</td>
</tr>
<tr>
<td>p2: if gate2 ≠ 0 goto p1</td>
<td>q2: if gate2 ≠ 0 goto q1</td>
</tr>
<tr>
<td>p3: gate2 ← p</td>
<td>q3: gate2 ← q</td>
</tr>
<tr>
<td>p4: if gate1 ≠ p</td>
<td>q4: if gate1 ≠ q</td>
</tr>
<tr>
<td>p5: if gate2 ≠ p goto p1</td>
<td>q5: if gate2 ≠ q goto q1</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>p6: gate2 ← 0</td>
<td>q6: gate2 ← 0</td>
</tr>
</tbody>
</table>

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

- 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

loop forever

<Diagram of algorithm>

Algorithm 5.6: Fast algorithm for two processes (2)

integer gate1 ← 0, gate2 ← 0

boolean wantp ← false, wantq ← false

loop forever

<Diagram of algorithm>

- 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: \textbf{await wantq=false} \quad \textbf{Pi}: \textbf{For all other j}

  \textbf{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 \neq i