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

Lesson 4

Propositional Calculus
• Implication
  \((A_1 \land A_2 \land \cdots \land A_n) \rightarrow B\)
  \(A \rightarrow B\)
  \(\text{Implication}
  \)

  • Premise or antecedent
  • Conclusion or consequent

• Formula
  • Atomic proposition
  • Atomic propositions or formulaes 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/false if \(v(f) = T\), false if \(v(f) = F\)

Methods for Proving Formulae Valid
• Induction proof \(F(n)\) for all \(n=1, 2, 3, \ldots\)
  • \(F(1)\)
  • \(F(n) \rightarrow F(n+1)\)

  • Dual approach: \(f\) is valid \(\iff\) \(f\) 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., \(v(f) = T\)
  • \(f\) is not valid
  • \(\neg f\) is valid

  • Proof by contradiction
  • Assume \(f\) is not valid
  • Deduce contradiction with propositional calculus
  \(\neg \forall X\)

Propositional Calculus
• Formula
  \((A_1 \land A_2 \land \cdots \land A_n) \rightarrow B\)
  \(A \rightarrow B\)
  \(\text{Implication}
  \)

  • Premise or antecedent
  • Conclusion or consequent

  • Atomic proposition
  • Atomic propositions or formulaes 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/false if \(v(f) = T\), false if \(v(f) = F\)

Propositional Calculus
• Formula
  \((A_1 \land A_2 \land \cdots \land A_n) \rightarrow B\)
  \(A \rightarrow B\)
  \(\text{Implication}
  \)

  • Premise or antecedent
  • Conclusion or consequent

  • Atomic proposition
  • Atomic propositions or formulaes 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/false if \(v(f) = T\), false if \(v(f) = F\)

Methods for Proving Formulae Valid
• Deductive proof
  • Deduce formula from axioms and existing valid formulaes
  • Start from the “beginning”

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

Lecture 4: Verifying Solutions and Turn-Ticket Problem
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(...)

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 (not covered in this course!)

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

Formulae

<table>
<thead>
<tr>
<th>Algorithm 3.8: Third attempt</th>
<th>boolean wantp = false, wantq = false</th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
<td>p₁: non-critical section</td>
<td>q₁: non-critical section</td>
</tr>
<tr>
<td>p₂: wantp = true</td>
<td>q₂: wantq = false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p₃: await wantp = false</td>
<td>q₃: await wantq = false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p₄: critical section</td>
<td>q₄: critical section</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p₅: wantp = false</td>
<td>q₅: wantq = false</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 only statements that affect propositions in invariant

Mutex Proof

- Invariant
  - Can not prove directly (yet) – too difficult
- Need proven Lemma 4.3
  - Lemma 4.3: p₃.5 → wantp is invariant
  - Lemma 2: wantp → p₃.5 is invariant
  - Lemma 4: p₃.5 → wantp and q₃.5 → wantq are invariants
  - Proof not covered here
- Can now prove original invariant (p₄ ∨ q₄)
  - Inductive proof with Lemma 4.3
  - Details on next slide
Lecture 4: Verifying Solutions and Turn-Ticket Problem

**Concurrency Programming (RIO) 24.1.2011**

**Problem 3**

**Mutex Proof**

- **Lemma 4.3**: If \( p_3 \) and \( q_3 \) are invariant, then...
- **Theorem 4.4**: \( \neg(p_4 \land q_4) \) is invariant.

- **Proof**:
  - Initial state: trivial
  - Only states \( \{p_3, ..., \} \) need to be considered
  - \( p_4 \) may become true only here, i.e., state \( \{p_4, q?, ..., \} \)
  - States \( \{..., q_3, ..., \} \) are similar, symmetrical
  - Can execute \( \{p_3, ..., \} \) if \( \neg wantq \)
  - Because \( \neg wantq \), \( q_4 \) is also false (Lemma 4.3)
  - Next state cannot be \( \{p_4, q_4, ..., \} \), i.e., \( \neg(p_4 \land q_4) \)

**Temporal Logic**

- **Propositional logic with extra temporal operators**
- **Computation**
  - Infinite sequence of states: \( S_0, S_1, S_2, ... \)
- **Temporal operators**
  - Value (T or F) of given predicate does not necessarily depend only on current state
  - Initial state: trivial
  - Only states \( \{p_3, ..., \} \) need to be considered
- **Examples**
  - Mutex must always be true
  - Eventually or diamond (\( \diamond \)) operator
    - \( p \) true in state \( s_i \), if \( p \) true in some \( s_j \), \( j \geq i \)
    - Example: no starvation means that something eventually will become true

**Other Temporal Logic Operators**

- **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**
  - \( p U q \) true in state \( s_i \), if \( p \) true in every state in future until \( q \) becomes true
- **Not used (needed) in this course...**

**Some Laws of Temporal Logic**

- **DeMorgan**
  - \( \neg(A \lor B) \iff (\neg A \land \neg B) \)
  - \( \neg(A \land B) \iff (\neg A \lor \neg B) \)
- **Distributive Laws**
  - \( (A B) \iff (A V B) \iff (A \lor B) \)
- **Duality**
  - Not always is equivalent to eventually not
  - Not eventually is equivalent to always not

**Sequence**

- **Eventually always**
  - Will come true and then stays true forever
- **Always eventually**
  - 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

Concurrent Programming (RIO) 24.1.2011

Lecture 4: Verifying Solutions and Turn-Ticket Problem

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)

| p            | q
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( q &lt; p )</td>
<td>( q &lt; p )</td>
</tr>
<tr>
<td>( q &gt; p )</td>
<td>( q &gt; p )</td>
</tr>
</tbody>
</table>

In real life always use atomic!

- 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

Temporal logic

Alg. 5.1

Bakery for n Processes

Algorithm 5.2: Bakery algorithm (N processes)

| p | q
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( p = \text{number}[0] + 1 )</td>
<td>( q = \text{number}[i] ) for all ( i )</td>
</tr>
<tr>
<td>( q = \text{number}[i] )</td>
<td>( p = \text{number}[0] ) for all ( i )</td>
</tr>
</tbody>
</table>

- No write competition to shared variables
- 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[i] 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.

  \[ \text{e.g., max 100 processes, CS less than 0.01\% of executed code} \]

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 5.3: Bakery Algorithm (without atomic assignment) (3)

Algorithm 5.4: Fast algorithm for two processes (outline)

- 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)

- 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
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

  Algorithm 5.6: Fast algorithm for two processes (outline) (2)

  \[
  \begin{align*}
  \text{integer} & \quad \text{gate1} = 0, \text{gate2} = 0 \\
  \text{loop} & \quad \text{forever} \\
  \text{if} & \quad \text{gate1} = 0 \\
  \text{if} & \quad \text{gate2} = 0 \\
  \text{if} & \quad \text{gate1} \neq p \\
  \text{if} & \quad \text{gate2} \neq q \\
  \text{P:} & \quad \text{wait} = \text{false} \\
  \text{Pi:} & \quad \text{for all other} j \\
  \text{await} & \quad \text{want}[j] = \text{false} \\
  \text{P last at gate1} & \quad \text{gate1} = q \\
  \text{Q last at gate2} & \quad \text{gate2} = 0
  \end{align*}
\]

- 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