Propositional Calculus

- Atomic propositions
  - A, B, C, ...
  - True (T) or False (F)

- Operators
  - not: \( \neg \)
  - disjunction, or: \( A \lor B \)
  - conjunction, and: \( A \land B \)
  - implication: \( A \rightarrow B \)
  - equivalence: \( A \equiv B \)

- Premise or antecedent
- Conclusion or consequent

- Formula:
  - Atomic proposition
  - Atomic propositions or formulas 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 \( v(f) = T \)
  - Formula \( f \) is 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 if \( \neg f \) is unsatisfiable
  - Find one interpretation that makes \( f \) true
  - Go through (automatically) all interpretations of \( \neg f \)
  - If such interpretation found, \( f \) is satisfiable, i.e., \( \neg f \) is false
  - Otherwise, \( f \) is valid

- Proof by contradiction
  - Assume: \( f \) is not valid
  - Conclude contradiction with propositional calculus
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 (‘Halt 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
    - malin tarkastin

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

Algorithm 3.8: Third attempt

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

Mutex Proof

Algorithm 3.8: Third attempt

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

Algorithm 3.8: Third attempt

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

- Invariant (p4 ∧ q4)
  - Can not prove directly (yet) - too difficult
- Need proven Lemma 4.3
  - Lemma 4.1: p3.3 → wantp is invariant
  - Lemma 4.2: wantp → p3.5 is invariant
  - Lemma 4.3: p3.5 → wantp and q3.5 → wantq are invariants
- 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

Mutex Proof

Lemma 4.3: \( p_{3..5} \) \( \rightarrow \) \( q_{3..5} \) invariants

Theorem 4.4: \( \neg (p_4 \lor q_4) \) is invariant

1. Prove \( (p_4 \lor q_4) \) inductively false in every state
2. Initial state: trivial
3. Only states \( \{p_3, \ldots\} \) need to be considered
4. \( p_4 \) may become true only here, i.e., states \( \{p_4, q?, \ldots\} \)
5. States \( \{\ldots, q_3, \ldots\} \) similar, symmetrical
6. Can execute \( \{p_3, \ldots\} \) only if \( \neg \)wantq = false (i.e., \( \neg \)wantq)
7. Because \( \neg \)wantq = false, \( q_4 \) is also false (Lemma 4.3)
8. Next state cannot be \( \{p_4, q_4, \ldots\} \), i.e., \( (p_4 \lor q_4) \) 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
- Always or box (\( \mathcal{F} \)) operator
  - \( \mathcal{F} \) a true in state \( s_i \) if \( \mathcal{F} \) a true in all \( s_j \), \( j \geq i \)
  - E.g., mutex must always be true
- Eventually or diamond (\( \mathcal{G} \)) operator
  - \( \mathcal{G} \) a true in state \( s_i \) if \( \mathcal{G} \) 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 \( s_i \) 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.

Sequence

- Eventually always
  - Will come true and then stays true forever
- Always eventually
  - Always will become true some times in future (again)

Some Laws of Temporal Logic

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

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
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)

- 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)
- q in non-critical section

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

- 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?

Advanced Critical Section Solutions

Ch 5 [BenA 06] (no proofs)

Bakery Algorithm
Bakery for N processes
Fast for N processes
Bakery for n Processes

- Mutex OK?
  - Yes, because of priorities at competition time
- Deadlock OK?
  - Yes, because of priorities at competition time
- Starvation OK?
  - 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?

Alg. 5.2

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 (5)

```java
loop forever
    p1: non-critical section
    p2: if (gate2 != 0) goto p1
    p3: if gate2 == p
    p4: if gate2 != p
    p5: critical section
    p6: gate2 = 0

loop forever
    q1: non-critical section
    q2: if gate2 != 0
    q3: if gate2 != q
    q4: if gate2 != q
    q5: critical section
    q6: gate2 = 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)

```java
loop forever
    p1: non-critical section
    p2: if gate2 != 0 goto p1
    p3: if gate2 == p
    p4: if gate2 != p
    p5: critical section
    p6: gate2 = 0

loop forever
    q1: non-critical section
    q2: if gate2 != 0
    q3: if gate2 != q
    q4: if gate2 != q
    q5: critical section
    q6: gate2 = 0
```

• 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
### Fast N Process Baker

- **Expand Alg. 5.6**
  - Still with just 2 gates

- **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

- **Algorithm 5.6**: Fast algorithm for two processes (2)

```plaintext
P

1. p1: gate1 ← p
2. p2: if gate2 ≠ 0 goto p1
3. p3: gate2 ← p
4. p4: if gate1 ≠ p goto q5  
      if gate2 ≠ q goto q3
5. p5: if gate2 ≠ p goto p1
6. p6: gate2 ← 0

Q

1. q1: gate1 ← q
2. q2: if gate2 ≠ 0 goto q1
3. q3: gate2 ← q
4. q4: if gate1 ≠ q goto q5
5. q5: if gate2 ≠ q goto q3
6. q6: gate2 ← 0
```

- P last at gate1
- Q last at gate 2
- P & Q: P advances, Q advances, i.e., no mutex (ouch!)

- • 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!)