Concurrent Programming (RIO) 18.1.2012

Lecture 4: Verifying Solutions and Turn-Ticket Problem

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
- Formula \((A_1 \land A_2 \land \cdots \land A_n) \rightarrow B\)
- Implication
  - Premise or antecedent
  - Conclusion or consequent
- Formula is true/false if its interpretation \(v(f)\) is true/false
- Given assignment values for each argument
- Formula is valid if it is a tautology
- Always true for all interpretations (all atomic propos. values)
- Formula is satisfiable if true in some interpretation
- Formula is falsifiable if sometimes false
- Formula is unsatisfiable if always false

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., \(f\) is not valid
    - Otherwise \(f\) is valid
- Proof by contradiction
  - Assume: \(f\) is not valid
  - Deduce contradiction with propositional calculus

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

Boolean algebra

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 \(p \rightarrow q\)
  - Can show that \(\neg(p \rightarrow q)\) cannot be true (or cannot become)
    - \(v(p) = T\) and \(v(q) = F\)
    - \(v(p) = q = T\) and \(v(p) \neq F\)
    - \(v(p) = q = F\) and \(v(p) \neq T\)
    - \(v(p) = F\) and \(v(q) \neq T\)

18.1.2012 Copyright Teemu Kerola 2012
**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 $A(x_1, x_2, \ldots)$ for input values $(x_1, x_2, \ldots)$
- postconditions $B(y_1, y_2, \ldots)$ for output values $(y_1, y_2, \ldots)$

Partial and total correctness with respect to $A(\ldots)$ and $B(\ldots)$

**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 $\neq 2"$ 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

**Mutex Proof**

- **Invariant** $\neg(p4 \land q4)$
  - True if mutex is broken
  - Mutex condition can be defined: $\neg(p4 \land q4)$
- **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** $\neg(p4 \land q4)$
  - 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

**Formulae**

- **Formula**: $p1 A q1 A \neg wantp A \neg wantq$
  - True only in the starting state
- **Formula**: $p4 A q4$
  - 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**

**Mutex Proof**

- **Lemma 4.3**
  - Need proven Lemma 4.3
  - Inductive proof with Lemma 4.3
  - Details on next slide
Mutex

Proof

Lemma 4.3: \( p_3 \ldots 5 \) \( \Rightarrow \) wantp and \( q_3 \ldots 5 \) \( \Rightarrow \) wantq invariants

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

- Prove \( (p_4 \vee q_4) \) 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., \( \{p_4, q_?, \ldots\} \)
  - States \( \{\ldots, q_3, \ldots\} \) similar, symmetrical
  - Can execute \( \{p_3, \ldots\} \) only if wantq=false, i.e., \( \neg\) wantq
  - Because wantq=false, \( q_4 \) is also false (Lemma 4.3)
  - Next state cannot be \( \{p_4, q_4, \ldots\} \), i.e., \( (p_4 \vee q_4) \) 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 (\( \Box \)) operator
  - A true in state \( s_i \) if A true in all \( s_j \), \( i \leq j \)\n- Eventually or diamond (\( \Diamond \)) operator
  - A true in state \( s_i \) if A true in some \( s_j \), \( i \leq j \)
  - 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 \( s_{i+2} \)
- Until eventually (U) operator
  - U p 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 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

Some Laws of Temporal Logic

- deMorgan
  - \( \neg(A \cap B) \) \( \Rightarrow \) \( (\neg A \vee \neg B) \)
  - \( \neg(A \vee B) \) \( \Rightarrow \) \( (\neg A \cap \neg B) \)

- Distributive Laws
  - \( (A \vee B) \cap (A \vee C) \) \( \Rightarrow \) \( (A \vee (B \cap C)) \)
  - \( (A \cap B) \vee (A \cap C) \) \( \Rightarrow \) \( (A \cap (B \vee C)) \)

- Duality
  - Not always is equivalent to eventually not
  - \( \neg A \Rightarrow \Diamond \neg A \) - Not eventually is equivalent to always not
  - \( A \Rightarrow \Box A \)
Advanced Critical Section Solutions
Ch 5 [BenA 06] (no proofs)

Bakery Algorithm
Bakery for N processes
Fast for N processes

Bakery Algorithm (2 processes)
Algorithm 5.1: Bakery algorithm (two processes)

\begin{align*}
\text{loop for } & p: \\
\text{loop for } & q:
\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)

Bakery Algorithm (n processes)
Algorithm 5.2: Bakery algorithm (N processes)

\begin{align*}
\text{loop for } & p: \\
\text{loop for } & q:
\end{align*}

- No write competition to shared variables
- Ticket numbers increase continuously while critical section is taken – danger?
- All other processes polled
  - Not so good!
Lecture 4: Verifying Solutions and Turn-Ticket Problem

### 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.

### 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)

```plaintext
loop forever
  integer gate1 = 0, gate2 = 0
  p:
    if gate2 != 0 goto p1
    if gate1 != p goto q1
    gate1 = p
    if gate1 != q goto q1
  q:
    if gate2 != q goto q1
    gate2 = q
```

- **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
Lecture 4: Verifying Solutions and Turn-Ticket Problem

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

Concurrent Programming (RIO) 18.1.2012

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

```
integer gate1 = 0, gate2 = 0

P = loop forever
    non-critical section
    if gate1 ≠ 0 goto pl
    else goto q1
q = loop forever
    non-critical section
    if gate2 ≠ 0 goto q1
    else goto q2
pl:
    if gate1 ≠ p
        goto pl
    else
        if gate2 ≠ q
            goto q1
        else
            if gate1 ≠ q
                goto q1
            else
                gate2 = 0
q1:
    if gate1 ≠ p
        goto pl
    else
        if gate2 ≠ q
            goto q1
        else
            gate2 = 0
```

- 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!)

Algorithm 5.6: Fast algorithm for two processes (2)

```
integer gate1 = 0, gate2 = 0
boolean wantp = false, wantq = false

P:
    if gate2 ≠ 0
        wantp = false
        goto pi
    else
        gate2 = p
    if gate1 ≠ p
        wantp = false
        goto pl
    else
        if gate2 ≠ q
            wantp = false
            goto q1
        else
            if gate1 ≠ q
                wantp = false
                goto q1
            else
                Q blocks here
                wantp = false
    if gate2 ≠ p
        wantq = false
        goto q1
    else
        if gate2 ≠ q
            wantq = false
            else
                if gate2 ≠ q
                    wantq = false
                    else
                        critical section
                        if gate2 ≠ 0
                            wantq = false
                        else
                            critical section
                            if gate2 ≠ 0
                                wantq = false
```

- Discuss

Fast N Process Baker

- Expand Alg. 5.6
  - Still with just 2 gates

```
P: await wantq=false
    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