Shared Variable Proof Methods, Hardware-Assisted Critical Sections

Johannes Åman Pohjola
CSE, UNSW
Term 2 2022
In the last lecture we introduced the critical section problem, the four properties of critical section solutions, and some solutions for two processes.
In the last lecture we introduced the **critical section** problem, the four properties of critical section solutions, and some solutions for two processes.

We also introduced the **SPIN** model checking tool for rigorous analysis of candidate solutions.
Where we are at

In the last lecture we introduced the critical section problem, the four properties of critical section solutions, and some solutions for two processes.

We also introduced the SPIN model checking tool for rigorous analysis of candidate solutions.

In this lecture, we will introduce a formal proof method for verifying safety properties, and apply it to a new kind of critical section solution that relies on hardware support.
Definition

A transition diagram is a tuple \((L, T, s, t)\) where:

- \(L\) is a set of locations (program counter values).
- \(s \in L\) is an entry location.
- \(t \in L\) is an exit location.
- \(T\) is a set of transitions.

A transition is written as \(\ell_i \xrightarrow{g} \ell_j\) where:

- \(\ell_i\) and \(\ell_j\) are locations.
- \(g\) is a guard \(\Sigma \to B\).
- \(f\) is a state update \(\Sigma \to \Sigma\).

Transition Diagrams

\[
i \leftarrow 0; \quad s \leftarrow 0; \\
\text{while } i \neq N \text{ do} \\
\quad s \leftarrow s + i; \\
\quad i \leftarrow i + 1 \\
\text{od}
\]

\[
i \leftarrow 0; \quad s \leftarrow 0; \\
\text{while } i \neq N \text{ do} \\
\quad s \leftarrow s + i; \\
\quad i \leftarrow i + 1 \\
\text{od}
\]
Definition

A *transition diagram* is a tuple \((L, T, s, t)\) where:

- \(L\) is a set of *locations* (program counter values).
- \(s \in L\) is a *entry location*.
- \(t \in L\) is a *exit location*.
- \(T\) is a set of *transitions*.
Definition

A transition diagram is a tuple \((L, T, s, t)\) where:

- \(L\) is a set of locations (program counter values).
- \(s \in L\) is a entry location.
- \(t \in L\) is a exit location.
- \(T\) is a set of transitions.

A transition is written as \(\ell_i \xrightarrow{g;f} \ell_j\) where:

- \(\ell_i\) and \(\ell_j\) are locations.
- \(g\) is a guard \(\Sigma \to \mathbb{B}\).
- \(f\) is a state update \(\Sigma \to \Sigma\).
Definition

A transition diagram is a tuple \((L, T, s, t)\) where:

- \(L\) is a set of locations (program counter values).
- \(s \in L\) is an entry location.
- \(t \in L\) is an exit location.
- \(T\) is a set of transitions.

A transition is written as \(\ell_i \xrightarrow{g;f} \ell_j\) where:

- \(\ell_i\) and \(\ell_j\) are locations.
- \(g\) is a guard \(\Sigma \rightarrow \mathbb{B}\).
- \(f\) is a state update \(\Sigma \rightarrow \Sigma\).

\[
i \leftarrow 0; \\
s \leftarrow 0; \\
\textbf{while } i \neq N \textbf{ do} \\
    s \leftarrow s + i; \\
    i \leftarrow i + 1 \\
\textbf{od}
\]
Definition

A \textit{transition diagram} is a tuple \((L, T, s, t)\) where:

- \(L\) is a set of \textit{locations} (program counter values).
- \(s \in L\) is a \textit{entry location}.
- \(t \in L\) is a \textit{exit location}.
- \(T\) is a set of \textit{transitions}.

A \textit{transition} is written as \(\ell_i \xrightarrow{g;f} \ell_j\) where:

- \(\ell_i\) and \(\ell_j\) are locations.
- \(g\) is a \textit{guard} \(\Sigma \rightarrow \mathbb{B}\).
- \(f\) is a \textit{state update} \(\Sigma \rightarrow \Sigma\).

\begin{align*}
i &\leftarrow 0; \\
s &\leftarrow 0; \\
\text{while } i \neq N \text{ do} & \\
\quad & s \leftarrow s + i; \\
\quad & i \leftarrow i + 1 \\
\text{od}
\end{align*}
Transition Diagrams

Definition

A *transition diagram* is a tuple \((L, T, s, t)\) where:

- \(L\) is a set of *locations* (program counter values).
- \(s \in L\) is an *entry location*.
- \(t \in L\) is an *exit location*.
- \(T\) is a set of *transitions*.

A *transition* is written as \(\ell_i \xrightarrow{g;f} \ell_j\) where:

- \(\ell_i\) and \(\ell_j\) are locations.
- \(g\) is a *guard* \(\Sigma \rightarrow \mathbb{B}\).
- \(f\) is a *state update* \(\Sigma \rightarrow \Sigma\).

\[i \leftarrow 0;\]
\[s \leftarrow 0;\]
\[\textbf{while } i \neq N \textbf{ do}\]
\[s \leftarrow s + i;\]
\[i \leftarrow i + 1\]
\[\textbf{od}\]
**Definition**

A *transition diagram* is a tuple \((L, T, s, t)\) where:
- \(L\) is a set of *locations* (program counter values).
- \(s \in L\) is a *entry location*.
- \(t \in L\) is a *exit location*.
- \(T\) is a set of *transitions*.

A *transition* is written as \(\ell_i \xrightarrow{g; f} \ell_j\) where:
- \(\ell_i\) and \(\ell_j\) are locations.
- \(g\) is a *guard* \(\Sigma \rightarrow \mathbb{B}\).
- \(f\) is a *state update* \(\Sigma \rightarrow \Sigma\).

**Transition Diagrams**

\[
i \leftarrow 0;
s \leftarrow 0;
\text{while } i \neq N \text{ do}
\begin{align*}
s &\leftarrow s + i; \\
i &\leftarrow i + 1
\end{align*}
\text{od}
\]
Definition

A transition diagram is a tuple \((L, T, s, t)\) where:

- \(L\) is a set of locations (program counter values).
- \(s \in L\) is an entry location.
- \(t \in L\) is an exit location.
- \(T\) is a set of transitions.

A transition is written as \(\ell_i \xrightarrow{\Sigma \rightarrow} \ell_j\) where:

- \(\ell_i\) and \(\ell_j\) are locations.
- \(g\) is a guard \(\Sigma \rightarrow \mathbb{B}\)
- \(f\) is a state update \(\Sigma \rightarrow \Sigma\).

```
i \leftarrow 0;
s \leftarrow 0;
while \ i \neq N \ do
\quad s \leftarrow s + i;
\quad i \leftarrow i + 1
od
```
Definition

A **transition diagram** is a tuple \((L, T, s, t)\) where:

- \(L\) is a set of **locations** (program counter values).
- \(s \in L\) is an **entry location**.
- \(t \in L\) is an **exit location**.
- \(T\) is a set of **transitions**.

A **transition** is written as \(\ell_i \xrightarrow{g;f} \ell_j\) where:

- \(\ell_i\) and \(\ell_j\) are locations.
- \(g\) is a **guard** \(\Sigma \to \mathcal{B}\).
- \(f\) is a **state update** \(\Sigma \to \Sigma\).

\[
i \leftarrow 0;
s \leftarrow 0;
\textbf{while } i \neq N \textbf{ do }
\begin{align*}
s &\leftarrow s + i; \\
i &\leftarrow i + 1
\end{align*}
\textbf{od}
\]
Floyd Verification

Recall the definition of a Hoare triple for \textit{partial correctness}:

\[
\{ \varphi \} \ P \ \{ \psi \}
\]
Floyd Verification

Recall the definition of a Hoare triple for *partial correctness*:

$\{\varphi\} \ P \ \{\psi\}$

This states that if the program $P$ successfully executes from a starting state satisfying $\varphi$, the result state will satisfy $\psi$. 

Floyd Verification

Recall the definition of a Hoare triple for *partial correctness*:

\[
\{ \varphi \} \ P \ \{ \psi \}
\]

This states that if the program \( P \) successfully executes from a starting state satisfying \( \varphi \), the result state will satisfy \( \psi \). Observe that this is a *safety property*.

**Verifying Partial Correctness**

Given a transition diagram \((L, T, s, t)\):
Floyd Verification

Recall the definition of a Hoare triple for *partial correctness*:

\[ \{ \varphi \} \quad P \quad \{ \psi \} \]

This states that if the program \( P \) successfully executes from a starting state satisfying \( \varphi \), the result state will satisfy \( \psi \). Observe that this is a *safety property*.

**Verifying Partial Correctness**

Given a transition diagram \((L, T, s, t)\):

1. Associate with each location \( \ell \in L \) an assertion \( Q(\ell) : \Sigma \rightarrow \mathbb{B} \).
Floyd Verification

Recall the definition of a Hoare triple for partial correctness:

$$\{\varphi\} \ P \ {\psi}$$

This states that if the program $P$ successfully executes from a starting state satisfying $\varphi$, the result state will satisfy $\psi$. Observe that this is a safety property.

**Verifying Partial Correctness**

Given a transition diagram $(L, T, s, t)$:

1. Associate with each location $\ell \in L$ an assertion $Q(\ell) : \Sigma \rightarrow \mathbb{B}$.
2. Prove that this assertion network is inductive, that is: For each transition in $T$

   $$\ell_i \xrightarrow{g;f} \ell_j$$

   show that:

   $$Q(\ell_i) \land g \Rightarrow Q(\ell_j) \circ f$$
Recall the definition of a Hoare triple for *partial correctness*:

$\{ \varphi \} \frac{}{P} \{ \psi \}$

This states that if the program $P$ successfully executes from a starting state satisfying $\varphi$, the result state will satisfy $\psi$. Observe that this is a *safety property*.

### Verifying Partial Correctness

Given a transition diagram $(L, T, s, t)$:

1. Associate with each location $\ell \in L$ an assertion $Q(\ell) : \Sigma \to \mathbb{B}$.
2. Prove that this assertion network is *inductive*, that is: For each transition in $T$

   $\ell_i \xrightarrow{g;f} \ell_j$

   show that:

   $$Q(\ell_i) \land g \Rightarrow Q(\ell_j) \circ f$$

3. Show that $\varphi \Rightarrow Q(s)$ and $Q(t) \Rightarrow \psi$. 
Floyd Verification

Recall the definition of a Hoare triple for *partial correctness*:

$$\{\varphi\} \ P \ {\psi}$$

This states that if the program $P$ successfully executes from a starting state satisfying $\varphi$, the result state will satisfy $\psi$. Observe that this is a *safety property*.

### Verifying Partial Correctness

Given a transition diagram $(L, T, s, t)$:

1. Associate with each location $\ell \in L$ an assertion $Q(\ell) : \Sigma \to \mathbb{B}$.
2. Prove that this assertion network is *inductive*, that is: For each transition in $T$ $\ell_i \xrightarrow{g;f} \ell_j$ show that:

   $$Q(\ell_i) \land g \Rightarrow Q(\ell_j) \circ f$$

3. Show that $\varphi \Rightarrow Q(s)$ and $Q(t) \Rightarrow \psi$.

Johannes will now demonstrate on the previous example
Adding Concurrency

\[ i \neq N; n \leftarrow x \]

\[ x, i \leftarrow n + 1, i + 1 \]
Adding Concurrency

Transition Diagrams

Owicki-Gries Method

Invariants and Machine Instructions

Parallel Composition

Given two processes $P$ and $Q$ with transition diagrams ($L_P, T_P, s_P, t_P$) and ($L_Q, T_Q, s_Q, t_Q$), the parallel composition of $P$ and $Q$, written $P \parallel Q$ is defined as ($L, T, s, t$) where:

- $L = L_P \times L_Q$
- $s = s_P s_Q$
- $t = t_P t_Q$
- $p_i q_i \xrightarrow{f} p_j q_j$ if $p_i q_i \xrightarrow{f} p_j \in T_P$ and $q_i q_j \xrightarrow{f} q_j \in T_Q$
Adding Concurrency
Adding Concurrency

Parallel Composition

Given two processes $P$ and $Q$ with transition diagrams $(L_P, T_P, s_P, t_P)$ and $(L_Q, T_Q, s_Q, t_Q)$, the parallel composition of $P$ and $Q$, written $P \parallel Q$ is defined as $(L, T, s, t)$ where:

- $L = L_P \times L_Q$
- $s = s_P s_Q$
- $t = t_P t_Q$
- $p_i q_i \xrightarrow{g;f} p_j q_i \in T$ if $p_i \xrightarrow{g;f} p_j \in T_P$
- $p_i q_i \xrightarrow{g;f} p_i q_j \in T$ if $q_i \xrightarrow{g;f} q_j \in T_Q$
State Space Explosion

If we were SPIN, we would immediately begin exhaustively analysing this large diagram. But human brains don’t have that much storage space.
State Space Explosion

If we were SPIN, we would immediately begin exhaustively analysing this large diagram. But human brains don’t have that much storage space.

Problem

The number of locations and transitions grows exponentially as the number of processes increases.
State Space Explosion

If we were SPIN, we would immediately begin exhaustively analysing this large diagram. But human brains don’t have that much storage space.

Problem

The number of locations and transitions grows exponentially as the number of processes increases.

We can only use Floyd’s method directly on the parallel composition (product) diagram in the most basic examples.
State Space Explosion

If we were SPIN, we would immediately begin exhaustively analysing this large diagram. But human brains don’t have that much storage space.

**Problem**

The number of locations and transitions grows exponentially as the number of processes increases.

We can only use Floyd’s method directly on the parallel composition (product) diagram in the most basic examples.

**Susan Owicki’s solution**

Define inductive assertion networks for $P$ and $Q$ separately. By proving some non-interference properties derive an inductive network for $P \parallel Q$ automatically. This means we won’t have to draw that large product diagram!
Owicki-Gries Method

**Steps**

To show \{\varphi\} P \parallel Q \{\psi\}:

1. Define local assertion networks P and Q for both processes.
2. Show that they're inductive.
3. For each location \( p \in L_P \), show that \( P(p) \) is not falsified by any transition of \( Q \).
   That is, for each \( q.g.f \rightarrow q' \in T_Q \):
   \[ P(p) \land Q(q) \land g \Rightarrow P(p) \circ f \]
4. Vice versa for Q.
5. Show that \( \varphi \Rightarrow P(s_P) \land Q(s_Q) \) and \( P(t_P) \land Q(t_Q) \Rightarrow \psi \).
Owicki-Gries Method

Steps
To show \( \{\varphi\} P \parallel Q \{\psi\} \):

1. Define local assertion networks \( P \) and \( Q \) for both processes. Show that they’re inductive.
Owicki-Gries Method

To show $\{\varphi\} P \parallel Q \{\psi\}$:

1. Define local assertion networks $P$ and $Q$ for both processes. Show that they’re inductive.
2. For each location $p \in L_P$, show that $P(p)$ is not falsified by any transition of $Q$. That is, for each $q \xrightarrow{g;f} q' \in T_Q$: $P(p) \land Q(q) \land g \Rightarrow P(p) \circ f$

Steps
Owicki-Gries Method

Steps

To show $\{\varphi\} P \parallel Q \{\psi\}$:

1. Define local assertion networks $P$ and $Q$ for both processes. Show that they’re inductive.

2. For each location $p \in L_P$, show that $P(p)$ is not falsified by any transition of $Q$. That is, for each $q \xrightarrow{g;f} q' \in T_Q$:
   
   \[ P(p) \land Q(q) \land g \Rightarrow P(p) \circ f \]

3. Vice versa for $Q$. 

---

Diagram:

- Transition Diagrams
- Owicki-Gries Method
- Invariants and Machine Instructions

**Owicki-Gries Method**

1. $i \neq N; n \leftarrow x$
2. $x, i \leftarrow n + 1, i + 1$
3. $i = N$
4. $P(p)$
5. $Q(q)$
6. $P(p) \land Q(q) \land g \Rightarrow P(p) \circ f$

- $i \neq N; m \leftarrow x$
- $x, j \leftarrow m - 1, j + 1$
- $j = N$
- $Q(q)$
- $P(p) \land Q(q) \land g \Rightarrow P(p) \circ f$

---

**Steps**

1. Define local assertion networks $P$ and $Q$ for both processes. Show that they’re inductive.

2. For each location $p \in L_P$, show that $P(p)$ is not falsified by any transition of $Q$. That is, for each $q \xrightarrow{g;f} q' \in T_Q$:
   
   \[ P(p) \land Q(q) \land g \Rightarrow P(p) \circ f \]

3. Vice versa for $Q$. 

---

**Diagram**

- $i \neq N; n \leftarrow x$
- $x, i \leftarrow n + 1, i + 1$
- $i = N$
- $P(p)$
- $Q(q)$
- $P(p) \land Q(q) \land g \Rightarrow P(p) \circ f$

- $j \neq N; m \leftarrow x$
- $x, j \leftarrow m - 1, j + 1$
- $j = N$
- $Q(q)$
- $P(p) \land Q(q) \land g \Rightarrow P(p) \circ f$
Owicki-Gries Method

To show \( \{ \varphi \} P \parallel Q \{ \psi \} \):

1. Define local assertion networks \( P \) and \( Q \) for both processes. Show that they’re inductive.

2. For each location \( p \in L_P \), show that \( P(p) \) is not falsified by any transition of \( Q \). That is, for each \( q \xrightarrow{g;f} q' \in T_Q \):

\[
P(p) \land Q(q) \land g \Rightarrow P(p) \circ f
\]

3. Vice versa for \( Q \).

4. Show that \( \varphi \Rightarrow P(s_P) \land Q(s_Q) \) and \( P(t_P) \land Q(t_Q) \Rightarrow \psi \).
How does it help?

The Owicki-Gries method generalises to $n$ processes, by requiring more interference freedom obligations.
How does it help?

The Owicki-Gries method generalises to \( n \) processes, by requiring more interference freedom obligations.

**Derived Assertion Network**

The automatic assertion network we get for the parallel composition from the Owicki-Gries method is the conjunction of the local assertions at each of the component states.
The Owicki-Gries method generalises to $n$ processes, by requiring more interference freedom obligations.

**Derived Assertion Network**

The automatic assertion network we get for the parallel composition from the Owicki-Gries method is the conjunction of the local assertions at each of the component states.

Assume $k$ transitions and $m$ locations per process. For $m$ processes, Floyd’s method spawns $2 + n \cdot k \cdot m^{n-1}$ proof obligations!
How does it help?

The Owicki-Gries method generalises to $n$ processes, by requiring more interference freedom obligations.

**Derived Assertion Network**

The automatic assertion network we get for the parallel composition from the Owicki-Gries method is the conjunction of the local assertions at each of the component states.

Assume $k$ transitions and $m$ locations per process. For $m$ processes, Floyd’s method spawns $2 + n \cdot k \cdot m^{n-1}$ proof obligations!

Owicki-Gries reduces that to $2 + n \cdot k \cdot (1 + (n - 1) \cdot m)$ — merely quadratic in $n$. 
Proving Mutual Exclusion

The Owicki-Gries method can be used to prove properties like Mutual Exclusion.
Proving Mutual Exclusion

The Owicki-Gries method can be used to prove properties like Mutual Exclusion. **How?** Don’t have a postcondition. Instead make the assertions at the critical sections **contradictory** – so that they cannot be true simultaneously.
Proving Mutual Exclusion

The Owicki-Gries method can be used to prove properties like Mutual Exclusion.

**How?** Don’t have a postcondition. Instead make the assertions at the critical sections contradictory – so that they cannot be true simultaneously.

**Caution:** Ensure that each transition does not violate the limited critical reference rule!
Proving Mutual Exclusion

The Owicki-Gries method can be used to prove properties like Mutual Exclusion.

**How?** Don’t have a postcondition. Instead make the assertions at the critical sections contradictory – so that they cannot be true simultaneously.

**Caution:** Ensure that each transition does not violate the limited critical reference rule!

<table>
<thead>
<tr>
<th>Manna-Pnueli Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer wantp, wantq ← 0, 0</td>
</tr>
<tr>
<td><strong>forever do</strong></td>
</tr>
<tr>
<td>p₁</td>
</tr>
<tr>
<td>p₂</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>p₃</td>
</tr>
<tr>
<td>p₄</td>
</tr>
<tr>
<td>p₅</td>
</tr>
<tr>
<td><strong>forever do</strong></td>
</tr>
<tr>
<td>q₁</td>
</tr>
<tr>
<td>q₂</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>q₃</td>
</tr>
<tr>
<td>q₄</td>
</tr>
<tr>
<td>q₅</td>
</tr>
</tbody>
</table>

Note: The p₂ and q₂ steps are one atomic step!
Machine Instructions

What about if we had a single machine instruction to swap two values atomically, $XC$?
Machine Instructions

What about if we had a single machine instruction to swap two values atomically, $XC$?

<table>
<thead>
<tr>
<th></th>
<th>bit common $\leftarrow 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit tp $\leftarrow 0$</td>
<td></td>
</tr>
<tr>
<td>forever do</td>
<td></td>
</tr>
<tr>
<td>$p_1$ non-critical section</td>
<td>repeat</td>
</tr>
<tr>
<td>$p_2$ $XC(tp, common)$</td>
<td></td>
</tr>
<tr>
<td>$p_3$ until tp = 1</td>
<td></td>
</tr>
<tr>
<td>$p_4$ critical section</td>
<td></td>
</tr>
<tr>
<td>$p_5$ $XC(tp, common)$</td>
<td></td>
</tr>
<tr>
<td>bit tq $\leftarrow 0$</td>
<td></td>
</tr>
<tr>
<td>forever do</td>
<td></td>
</tr>
<tr>
<td>$q_1$ non-critical section</td>
<td>repeat</td>
</tr>
<tr>
<td>$q_2$ $XC(tq, common)$;</td>
<td></td>
</tr>
<tr>
<td>$q_3$ until tq = 1</td>
<td></td>
</tr>
<tr>
<td>$q_4$ critical section</td>
<td></td>
</tr>
<tr>
<td>$q_5$ $XC(tq, common)$</td>
<td></td>
</tr>
</tbody>
</table>
One Big Invariant

Imagine assertion network(s) where every assertion is the same: An invariant.
One Big Invariant

Imagine assertion network(s) where every assertion is the same: An invariant.

**Benefit:** We don’t need to prove interference freedom — the local verification conditions already show that the invariant is preserved.
One Big Invariant

Imagine assertion network(s) where every assertion is the same: An invariant. **Benefit:** We don’t need to prove interference freedom — the local verification conditions already show that the invariant is preserved.

**Example (Exchange-based Critical Section Solution)**

Using assertions about the program counters, we can craft an invariant for the XC example!
One Big Invariant

Imagine assertion network(s) where every assertion is the same: An invariant.

**Benefit:** We don’t need to prove interference freedom — the local verification conditions already show that the invariant is preserved.

**Example (Exchange-based Critical Section Solution)**

Using assertions about the program counters, we can craft an invariant for the XC example!

\[ I \equiv (\text{common} \oplus tp \oplus tq) = 1 \land (P@p_4 \Rightarrow tp = 1) \land (Q@q_4 \Rightarrow tq = 1) \land \neg(\text{common} = tp \land \text{common} = tq) \]

Where \( \oplus \) is xor. Note: \( I \) is false at \( p_4q_4 \). So if this invariant is preserved we have mutex.
One Big Invariant

Imagine assertion network(s) where every assertion is the same: An invariant. **Benefit:** We don’t need to prove interference freedom — the local verification conditions already show that the invariant is preserved.

**Example (Exchange-based Critical Section Solution)**

Using assertions about the program counters, we can craft an invariant for the XC example!

\[
\mathcal{I} \equiv (\text{common} \oplus tp \oplus tq) = 1 \land (P@p_4 \Rightarrow tp = 1) \land (Q@q_4 \Rightarrow tq = 1) \land \neg(\text{common} = tp \land \text{common} = tq)
\]

Where \(\oplus\) is xor. Note: \(\mathcal{I}\) is false at \(p_4q_4\). So if this invariant is preserved we have mutex.

Lets prove mutual exclusion for XC!
What now?

- You now have all you need to complete Assignment 0 (warm-up), due Monday the 20th.
- Next week: We will examine some more sophisticated critical section solutions for \( n \) processes.
- We may also learn about *semaphores*, time permitting!.