Axiomatic properties of concurrent programs

Why study theory?
- it reveals the "essence" of some aspect of computer science
- it helps to clarify our thinking about difficult problems
- it is the foundation for building programming tools
- it can establish the properties of systems that cannot be shown through testing or other informal means

Which theories?
- Owicki-Gries: proving assertions about parallel or concurrent programs
- Hoare: proving monitors
- Milner's Calculus of Communicating Systems (CCS): modeling the behavior of agents and proving the equivalence of two agents

The Language

Resources:
Definition: a resource is a named collection of program variables
Syntax: resource <name> ( <list of variables> );
Example: resource r(x, y, z);
Cobegin-coend:
Definition: a set of concurrently executed statements which end, as a set, only when all of the individual statements have completed.
Syntax: <resource> : cobegin S_1 || S_2 || ... || S_n coend
Example: resource r(x, y, z):
cobegin x = x + 1; y = 10; coend;

The Language

Condition Critical Regions:
Definition: a set of statement having mutually exclusive access to one or more a given condition holds among the variables of the resource(s).
Syntax: with <resource list> when <condition> do <statements>
Example:
resource r(x, y, z):
with r when x > y do
begin
  x = x + 1;
  y = y - 1;
end

Assertions

An axiom for statement S involves two assertions of the form:
precondition \[
\{ P \} S \{ Q \} \]
postcondition

partial correctness: if P is true before S and if S terminates then Q holds after S.

An assertion about a resource r is an invariant, written I(r), if the invariant is true:
- when the program begins
- during execution except within a "with r..."

Parallel Execution Axiom

If \[ \{ P_1 \} S_1 \{ Q_1 \}, \{ P_2 \} S_2 \{ Q_2 \}, \ldots, \{ P_n \} S_n \{ Q_n \} \]
no variable free in P_i or Q_i is changed in S_j (i != j) all variables in I(r) belong to resource r,

then \[ \{ P_1 \land P_2 \land \ldots \land P_n \land I(r) \} \]
resource r; cobegin S_1 || S_2 || ... || S_n coend;
\[ \{ Q_1 \land Q_2 \land \ldots \land Q_n \land I(r) \} \]
Critical Section Axiom

If

\{ I(r) \land P \land B \} \implies \{ I(r) \land Q \}

I(r) is the invariant of the cobegin statement for which
S is a process
no variable free in P and Q is changed in any other process
then

\{ P \} \text{ with } r \text{ when } B \text{ do } S \{ Q \}

An Example

\{( x = 0 )\}
add1: begin y := 0; z := 0;
{ y = 0 \land z = 0 \land I(r) } 
resource r(x, y, z):
cobegin
{ y = 0 }
with r when true do
{ y = 0 \land I(r) }
begin x := x + 1; y := 1 end
{ y = 1 }
end
{ z = 0 }
with r when true do
{ z = 0 \land I(r) }
begin x := x + 1; z := 1 end
{ z = 1 }
end
I(z) = \{ x = y + z \}

A part of Hoare's bounded buffer monitor

bounded-buffer : monitor
begin ...
int count; condition nonfull, nonempty;
...
procedure append( x : portion )
begin
if count = N then nonfull.wait;
 // add element x to the buffer
count = count + 1;
nonempty.signal;
end
procedure remove( result x : portion )
begin
if count = 0 then nonempty.wait;
 // assign to x an element removed
 // from the buffer
count = count - 1;
nonfull.signal;
end
...count = 0; // initialization
end;

Dining Philosophers Example

See annotated code in Postscript file from class web pages first.
This code shows that:

\{ eating[i] = 0 \land I(forks) \} \implies \{ eating[i] = 0 \land I(forks) \}

where:
I(forks) = \{ 0 \leq eating[i] \leq 1 \land 
\begin{align*}
&\text{eating[i] = 1 } \implies \text{af[i] = 2 } \\
&\text{af[i] = 2 } \implies \text{eating[i+1] = 1} \\
&\text{for } 0 \leq i < 4 \\
\end{align*}

By the Parallel Execution Axiom this means:

\{ eating = 0 \land I(forks) \}
resource forks: cobegin DP_1 \parallel \ldots \parallel DP_4 coend
\{ eating = 0 \land I(forks) \}

The invariant, I(forks), can be used to prove other properties about the solution.

Dining Philosophers Example

Prove that two adjacent philosophers, DP_1 and DP_{i+1}, cannot both be eating concurrently.

First,

\begin{align*}
\text{DP}_1 \text{eating} &\implies \text{eating[i]} = 1 \land I(forks) \\
\text{DP}_{i+1} \text{eating} &\implies \text{eating[i+1]} = 1 \land I(forks) \\
\end{align*}

Together this means:

\begin{align*}
\text{eating[i]} = 1 \land \text{eating[i+1]} = 1 \land I(forks) \\
\implies \text{eating[i]} = 1 \land \text{eating[i+1]} = 1 \land I(forks) \\
\implies \text{eating[i]} = 1 \land \text{eating[i+1]} = 1 \land I(forks) \\
\implies \text{false} \\
\end{align*}

Hoare's Proof Rules for Monitors

With each monitor define an invariant I that is true:

- after initialization
- after each procedure assuming that it was true before each procedure
- before each wait operation

With each condition variable define an assertion B that is true before each signal operation on that condition variable and which may be assumed true after the corresponding wait.

\begin{align*}
I &\implies \{ \text{cond.wait} \} \land I \\
I &\land B \implies \{ \text{cond.signal} \} \land I \\
\end{align*}

Finding the "$\text{right}$" invariant and the "$\text{right}$" assertions on conditions may be difficult.
Part of the Proof

For the bounded buffer monitor -

invariant: \( I = 0 \leq \text{count} \leq N \)
nonempty: \( B = 0 < \text{count} \)
nonfull: \( B' = \text{count} < N \)

\{ I \}
procedures append( x : portion )
begin
\{ I \}
if count = N \( \text{true} \)
false then \{ I \} nonfull.wait; \{ I \ \! A \ B' \}
\{ I \ \! A \ B' => 0 <= \text{count} < N \}
\{ I \ \! A \! (\text{count} = N) => 0 <= \text{count} < N \}
\{ 0 <= \text{count} < N \}
count = count + 1;
\{ I \ A \ B \}
nonempty.signal;
end