Summary of Owicki-Gries method
for SE2011 in 2014s1
(revising material from CS6721 2012s1)
======================================

===1 Context and motivation

Assume we have a collection of straight-line programs P1..P_N so that each one (for simplicity) is just a sequence of assignment statements, one after the other. These programs are to be run in parallel, interleaving at the statement level: only one statement executes at a time, but we don't know in advance "which Pn will go next". (This is called "interleaving" concurrency, and is the usual assumption.) The only thing we know about -which- statement is next to be executed, at any time, is that the statements within any particular program (sometimes called a "thread") are executed in order: aside from that, the programs' statements can be interleaved arbitrarily.

Here's an example. Suppose you have a music playlist containing 3 audio books, each one divided into 3 chapters one after the other: 9 chapters in all. You can listen to them them in order (Book 1, then 2 then 3); or you can press "Shuffle" and listen to the books all mixed up together. An "intelligent" shuffle however would make sure that within each book the chapters came in the right order, even if other books’ chapters came in between. In that case there would be \( 9! / (3!3!3!) = 1,680 \) possible intelligent shuffles.

If you had three straight-line programs of 3 statements each, without any AWAIT statements (see below), then similarly there would be 1,680 possible orders in which the statements could be executed. If for a SE2011 assignment you were to check the correctness of that 3-process system by providing assertions for each possible sequence, then to get it done in two working-weeks of 8-hour days

You would have to check a new, slightly different program's correctness every 3 minutes of each working day.

There is a better way.

===2 Assertions for concurrent shared-variable programs

The method we use is named Owicki-Gries, after its discoverer Susan Owicki and her PhD supervisor David Gries.

The whole system has an "initialisation" assertion at the beginning: this is its (overall) precondition. At its end it has a "finalisation" assertion: this is its (overall) postcondition. What it is supposed to do (ie what we want to prove) is to ensure that if the whole system is started in a state satisfying its precondition then, when it has terminated (ie every one of its constituent programs has terminated) the postcondition will be true no matter how the statements have interleaved.

From now on we'll call the individual programs "processes" and the overall program will be the (concurrent) "system".

Each of the individual processes has an assertion

- a. before its first statement,
- b. between every pair of its statements and
- c. after its last statement;

and what we now describe is how those assertions are checked for validity, and how they collectively establish correctness of the system as a whole.

===3 First step: Initialisation and termination

The system's initialisation assertion must imply each process's initial assertion (a) above; and all the process's final assertions (c) taken together must imply the finalisation assertion of the system.

===4 Second step: Local correctness of processes' individual assertions

For every Hoare-triple

\[ \{ \text{precondition}\} \text{ statement } \{ \text{postcondition}\} \]

within individual processes we must prove Local Correctness, abbreviated LC, which is simply that it is a valid Hoare triple in the sense we understand already.
Third step: Global correctness of processes' individual assertions

For every local assertion in some process $P_n$ (thus not the system initialisation, and not the system finalisation), we must show

\[
\{\text{local assertion in } P_n \& \text{ pre-condition of statement elsewhere}\} \quad \text{statement elsewhere} \quad \{\text{that same local assertion in } P_n\}
\]

for every "statement elsewhere" in any -other- process except $P_n$. This is called Global Correctness, abbreviated GC.

Obviously we need only consider statements that assign to variables actually occurring in the local assertion. If they don't, then the GC condition is satisfied trivially.

--- 6 Simple example

\[
\begin{align*}
\{x=0\} & & \text{// initial} \\
\{x=0 \lor x=2\} & \quad \{x=0 \lor x=1\} & \text{// individual} \\
x:=x+1 & \quad x:=x+2 \\
\{x=1 \lor x=3\} & \quad \{x=2 \lor x=3\} & \text{// individual} \\
\{x=3\} & \text{// final}
\end{align*}
\]

--- 6.1 First step

\[
\begin{align*}
x=0 \Rightarrow x=0 & \lor x=2 \\
x=0 \Rightarrow x=0 & \lor x=1 \\
(x=1 & \lor x=3) \land (x=2 & \lor x=3) \Rightarrow x=3
\end{align*}
\]

--- 6.2 Second step

\[
\begin{align*}
\{x=0 \lor x=2\} & \quad x:=x+1 \quad \{x=1 \lor x=3\} \\
\{x=0 \lor x=1\} & \quad x:=x+2 \quad \{x=2 \lor x=3\}
\end{align*}
\]

--- 6.3 Third step

\[
\begin{align*}
\{(x=0 \lor x=2) & \land (x=0 \lor x=1)\} \\
\{x=0\} & \\
x:=x+2 \\
\{x=2\} & \\
\{x=0 \lor x=2\}
\end{align*}
\]

and three more like that...

--- 7 Example: Peterson's algorithm for mutual exclusion

These 4 lines of code (symmetric on the two sides, thus 8 lines in all over the two threads A,B) took 15 years to discover (mid 1960's to early 1980's).

```plaintext
VAR a,b: Boolean:= FALSE,FALSE;
VAR t: {A,B} // uninitialised

{ !a && !b }
Other code of A
a:= true
\text{Critical section A}
\{a\} Critical section A
a:= false

Other code of B
b:= true
\text{Critical section B}
b:= false
```

We have to show that the two critical sections can never be active simultaneously.

--- 7.1 Add some Hoare-assertions

```plaintext
{ !a && !b }
Other code of A
a:= true
\text{Critical section A}
\{a\} Critical section A
a:= false

Other code of B
b:= true
\text{Critical section B}
b:= false
```

We add \{a\} and \{b\} as preconditions of the critical sections. Since neither "a" nor "b" is assigned-to by the other process, we can see trivially that it is both LC and GC to place the assertions there.
Proving invariance of !(a && b) would be enough to establish mutual exclusion. But in fact we cannot prove invariance of !(a && b), because it is not actually invariant! Consider the situation where each of the two processes has executed just its first two steps.

We need more.

---7.2 Add assertions derived from the AWAIT statements

We add assertions {!b || t==B} and {!a || t==A } as preconditions of the critical sections.

```plaintext
{ !a && !b }

<table>
<thead>
<tr>
<th>Other code of A</th>
<th>Other code of B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:= true</td>
<td>b:= true</td>
</tr>
<tr>
<td>t:= A</td>
<td>t:= B</td>
</tr>
<tr>
<td>AWAIT !b</td>
<td></td>
</tr>
<tr>
<td>{a} {!b</td>
<td></td>
</tr>
</tbody>
</table>

Critical section A

<table>
<thead>
<tr>
<th>Critical section B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:= false</td>
</tr>
</tbody>
</table>
```

These are trivially LC because of the AWAIT statements immediately before them. But for these we do have to check global correctness, since b,t are used in A but assigned-to in B, and vice versa. We can however now see our correctness condition, since

\[
a && (!b || t==B) && b && (!a || t==A) \]

\[
= a && (false || t==B) && b && (false || t==A) \\
= a && t==B && b && t==A \\
= false ,
\]

thus showing the two annotations on the critical sections are inconsistent. Therefore if we succeed in establishing LC and GC for these assertions, we will have shown that the two critical sections cannot be simultaneously active.

---7.3 Local correctness

It's trivial.

---7.4 Global correctness

The GC of {a} and {b} are trivial, as we already remarked. For the other two, by symmetry we need only check {!b || t==B} for GC --- which means we must go through all the statements in Thread B that assign either to b or t and make sure they cannot falsify {!b || t==B} in Thread A.

Thus we assume {!b || t==B} holds. Any statement (in B) falsifying it must either set b to true or t to A. But B never sets t to A, so we need only examine b:= true in B. How do we know it can't falsify b when A is in its critical section?

---7.5 Add even more assertions

```plaintext
{ !a && !b }

<table>
<thead>
<tr>
<th>Other code of A</th>
<th>Other code of B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:= true</td>
<td>b:= true</td>
</tr>
<tr>
<td>t:= A</td>
<td>t:= B</td>
</tr>
<tr>
<td>AWAIT !b</td>
<td></td>
</tr>
<tr>
<td>{a} {!b</td>
<td></td>
</tr>
</tbody>
</table>

Critical section A

<table>
<thead>
<tr>
<th>Critical section B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:= false</td>
</tr>
</tbody>
</table>
```

Unfortunately the Hoare triple {!b || t==B} b:= true {!b || t==B} is not valid, and so we don't have GC trivially. But in a sense we nearly have it, because the very next statement t:= B in B does re-establish {!b || t==B}, suggesting that if we made the two statements atomic together, ie if we combined the b:= true; t:= B into the single b,t:= true,B, we would have GC. And indeed we do.

Thus this "near miss" in the verification focusses out attention on the fact that each processes main vulnerability is between those two statements. It is encouraging also because we know that in the case of both processes being in their critical sections, perforce neither one can be in that vulnerable location.

We add (effectively) a pair of local, auxiliary Boolean variables α, β that are true just at
the point \( t := A \) resp \( t := B \) are about to be executed, and we abbreviate their use by labelling that statement with their names. Thus we have now

\[
\{ !a \land !b \}
\]

Other code of A
\[
\begin{align*}
\alpha: & \ t := A \\
\text{AWAIT}: & \ !b \lor t==B \\
\{a\} & \{!b \mid t==B \mid \beta\} \quad \text{Critical section A} \\
\end{align*}
\]

a:= false

Other code of B
\[
\begin{align*}
\beta: & \ t := B \\
\text{AWAIT}: & \ !a \lor t==A \\
\{b\} & \{!a \mid t==A \} \quad \text{Critical section B} \\
\end{align*}
\]


where in general \( x := X; \alpha := Y \) abbreviates \( x, \alpha := X, \text{true}; y, \alpha := Y, \text{false} \) and we assume an initialisation \( \{!0\} \). We use those auxiliary variables to capture the fact that the processes are, or are not, at those locations. Thus we have

\[
\{ !a \land !b \}
\]

Other code of A
\[
\begin{align*}
\alpha: & \ t := A \\
\text{AWAIT}: & \ !b \lor t==B \\
\{a\} & \{!b \mid t==B \mid \beta\} \quad \text{Critical section A} \\
\end{align*}
\]

\[a:= \text{false}\]

Other code of B
\[
\begin{align*}
\beta: & \ t := B \\
\text{AWAIT}: & \ !a \lor t==A \\
\{b\} & \{!a \mid t==A \mid \alpha\} \quad \text{Critical section B} \\
\end{align*}
\]

\[b:= \text{false}\]

The LC for these is trivial. For the GC we note that the assignment \( b := \text{true} \) that falsifies \( !b \) at the same time makes \( \beta \) true; and the assignment \( \beta: t := B \) that makes \( \beta \) false makes \( t==B \) true.

We must finish off by re-checking that these new assertions are still inconsistent (which they must be, to establish mutual exclusion). We have

\[
\begin{align*}
\text{a \land !a \land (} & !b \mid t==B \mid \beta \text{) \land b \land !b \land (} !a \mid t==A \mid \alpha \text{)} \\
\Rightarrow & \ t==B \land t==A \\
\Rightarrow & \ \text{false}
\end{align*}
\]
as before.

*** Postscript: Liveness

The really remarkable thing about Peterson’s algorithm, and the reason it took 15 years to discover (*), is that it guarantees fair treatment of A and B as well as protecting their critical sections from mutual destructive interference.

That is, beyond the system’s being partially correct (never gets the wrong answer), it is guaranteed that the system can never get stuck: we can be sure that if A (or B) “wants” to enter the critical section then eventually it will, no matter what B (or A) does.

One example of “getting stuck” is deadlock, where each thread is waiting for the other. Absence of deadlock is easy to see in this case: it occurs only when the two processes are both stuck in their AWAIT statements, and for that to happen both AWAIT conditions must be false at the same time. We calculate

\[
\begin{align*}
\neg ( & !b \mid t==B) \land \neg (!a \mid t==A) \\
\Rightarrow & \ b \land t==A \land a \land t==B \\
\Rightarrow & \ t==A \land t==B \\
\Rightarrow & \ \text{false}
\end{align*}
\]

so establishing absence of deadlock.

But how do we show for example that A is never perpetually ignored in favour of B? That’s a question of fair treatment, of “liveness” in this case --- and for that we need a -variant- argument... :-)

Carroll Morgan
29 May 2014

(*) To write 4 correct lines of code in 15 years is a coding rate of roughly 1 LoC every 4 years.