How to write an invariant proof

Mark Greenstreet

November 17, 2013

1 Peterson's Algorithm

I'll use Peterson's mutual exclusion algorithm as an example through much of this document. Figure 1 shows the code. I assume that there are two clients that use the lock. Each client performs the operations:

```
C1:  int my_id = ...; % 0 or 1
C2:  bool done = ...; % set by the non-critical code
C3:  non-critical code;
C4:  while(not done) {
C5:    lock(my_id);
C6:      critical section;
C7:    unlock(my_id);
C8:    non-critical code;
C9:  }
```

In English, each client starts in its not critical code. Any number of times (including zero), it may request the lock; execute code in a critical section once it has the lock; release the lock; and execute code that doesn't require protection by the lock. To show that Peterson's algorithm guarantees mutual exclusion, we want to prove that the program can never enter a state where both clients are in their critical sections at the same time.

Let `PC[`i`]` indicate the current "program counter" of thread i. To show mutual exclusion, we what to show that

$$\neg((\texttt{PC[0]} = \texttt{C6}) \wedge (\texttt{PC[1]} = \texttt{C6}))$$

holds at all times.

```
P01: % shared variables:
P02: bool flag[2] = {false, false};
P03: int victim = 0;
P04:
P05: lock(myId) {
P06:   int otherId = 1 - myId;% know your neighbour
P07:   flag[myId] = true;     % express intent to lock
P08:   victim = myId;         % you go first, please
P09:   while(flag[otherId] && (victim == myId)); % spin
P10: }
P11:
P12: unlock(myId) {
P13:   flag[myId] = false;
P14: }
```

Figure 1: Peterson's Mutual Exclsusion Algorithm

2 Modes

With 14 lines of code for Peterson's algorithm and 9 for the clients, we could try to reason about all 23 possible PC values for both thread, a total of 529 combinations. I'll simplify this by identifying four "modes" for each thread.

- **What's a mode?** It's a name given to a range of PC values.
- **How do we pick these ranges?** Look for "significant events." For example, entering or leaving the critical region and changes to shared variables that are used by the mutual exclusion algorithm.

For this example, I'll identify the following modes:

Idle: The thread doesn't have the lock nor is it trying to acquire it. This is:

`PC[`i`]` $\in \{$`C0`$, \ldots$ `C5`$\}$, before requesting the lock);

`PC[`i`]` $\in \{$`P05`, `P06`$\}$, in the `lock()` function but the thread hasn't changed any global variables yet;

`PC[`i`]` $=$ `P14`, in the `unlock()` function and done changing global variables;

`PC[`i`]` $\in \{$`C8`, `C9`$\}$, in the client, after the return from `unlock()` and before the next call (if any) to `lock()`.

Enter: `PC[`i`]` $=$ `P08`, the thread has indicated its intention to acquire the lock by setting `flag[`i`]`, but it hasn't set `victim` yet.

Spin: `PC[`i`]` $=$ `P09`, the thread is spinning, waiting to enter its critical region.

Crit: the thread is in its critical region,

`PC[`i`]` $= \{$`P10`$, \ldots$ `P13`$\}$, returning from `lock()` or just called `unlock()` but the thread hasn't changed any global variables yet.

`PC[`i`]` $= \{$`C6`, `C7`$\}$, the client is in its critical section, or is ready to call `unlock()`.

Let $mode(i)$ denote the "mode" (Idle, Enter, Spin, or Crit) of thread i. Now we can state the mutual exclusion property as:

$$\neg((mode(0) = \text{Crit}) \wedge (mode(1) = \text{Crit}))$$

While this property holds initially, when both threads are in mode "Idle", how can we show that it holds throughout the execution?

3 Reachable states

I'll describe the state of the program with a triple $\{mode,$ `flag`, `victim`$\}$. The initial state of the program is:

$$\{mode, \texttt{flag}, \texttt{victim}\} \quad = \quad \{[\text{Idle}, \text{Idle}], [\texttt{false}, \texttt{false}], 0\}$$

A state, s, is *reachable* if s is the initial state or if there is a state r that has been shown to be reachable, and s is reachable from r by performing one step of thread 0 or thread 1. I'm being somewhat informal in this explanation because I'm using the modes rather than the detailed PC, but it makes the examples **way** simpler.

For example, from then initial state, we could perform actions of thread 0 until it modifies at least one of mode, `flag`, or `victim`. This means that thread 0 completes its non-critical section, calls `lock()` and reaches line `P08` where it is in mode "Enter" with `flag[0]` $=$ `true`. Thus we conclude that state

$$\{[\text{Enter}, \text{Idle}], [\texttt{true}, \texttt{false}], 0\}$$

Conversely, we could have considered an execution where thread 1 reaches line `P08` first, and obtain the state

$$\{[\text{Idle}, \text{Enter}], [\texttt{false}, \texttt{true}], 0\}$$

From each of these states, we could explore what other states they can reach by performing actions of thread 0 or thread 1. Because the state consists of two mode variables (each of which can take one of four values), two `flag` variables (each of which can take one of two values), and the `victim` variable (that must be either 0 or 1), there are at most $(4 \cdot 4) \cdot (2 \cdot 2) \cdot 2 = 128$ reachable states. So, this state exploration will terminate, and we could determine whether or not the program can reach a state where both threads are in mode "Crit", thus violating mutual exclusion. This is a very tedious way to verify a program, and it's impractical for a program with even a few more variables.

4 Invariants

Invariants give us a way of proving properties of all reachable states of a program without having to enumerate these states one by one. Let I be a predicate over states. In other words, if s is a state (reachable or not) of the program, then $I(s)$ is either true or false – that's just saying that I is a predicate. Let s be any state of the program – we won't worry about whether or not s is reachable – as I illustrated above, it can be hard work to figure out whether or not a state is reachable – that's why we are using invariants! Let s' be any state that is reachable in one step from s. For our example with Peterson's algorithm, we can start with state s and ask: what happens if we perform one step of thread 0 starting from state s? What happens if we execute one step of thread 1?

We say that I is an invariant if for any states s and s' such that s' is reachable in one step from s, then if I holds in state s, i also must hold in state s'. If you prefer formulas, let S be the set of all states (reachable or not) of the program, and let $R \subseteq S \times S$ be the relation such that $(s, s') \in R$ iff the program can move from state s to state s' in one step of one of the threads.

$$IsInvariant(I) \quad \equiv \quad \forall (s, s') \in R.\ I(s) \Rightarrow I(s')$$

Of course, if you prefer English to math formulas, the sentence at the beginning of this paragraph says the same thing.

Here's the wonderful thing about invariants: if a program ever reaches a state that satisfies some invariant, I, then *all* subsequent states of the program will satisfy I – we just repeatedly apply the fact that I holding in one state guarantees that I will hold in the next state. Now, let's say we find an invariant I that holds in the *initial state* state of the program. Then, I holds in *all* reachable states of the program. Let Q be some property (such as mutual exclusion) that we want to have hold in all states of the program. If we can find an invariant I such that I holds in the initial state of the program, and I implies Q, then we know that Q must hold in all reachable states of the program. To verify that I is an invariant, we only need to reason about individual actions of the program. In particular, invariants spare us from having to reason about long sequences of states.

Why not just use Q as our invariant? Consider the mutual exclusion problem again. We have:

$$Q \quad = \quad \neg((mode(0) = \text{Crit}) \wedge (mode(1) = \text{Crit}))$$

Consider the state

$$s \quad = \quad \neg\{[\text{Crit, Spin}],\ [\texttt{false}, \texttt{false}],\ 0\}$$

Clearly, s satisfies Q. However, if we perform one step of thread 1, we reach the state

$$s' \quad = \quad \neg\{[\text{Crit, Crit}],\ [\texttt{false}, \texttt{false}],\ 0\}$$

which violates mutual exclusion – it doesn't satisfy Q. Does this mean that Peterson's algorithm is faulty? No. It just means that state s is not reachable by the program. How can we show that? We'll use an invariant.

5 Finding an invariant

Let's continue with the example above. Why do we believe that state s is unreachable? If we look at the Nov. 14 slides, we'll see that we found a relation between `flag[`i`]` and $mode(i)$:

$$\texttt{flag[}i\texttt{]} \quad = \quad mode(i) \in \{\text{Enter, Spin, Crit}\}$$

$$I_1 \quad \equiv \quad \forall i \in \{0,1\}.\ \texttt{flag[}i\texttt{]} = (mode(i) \in \{\text{Enter, Spin, Crit}\})$$

We can easily show that I_1 is an invariant. We do this by considering each action of the program:

Thread i makes a transition from "Idle" to "Enter": This occurs at line `P07` which sets `flag` to true. Thus $mode(i) =$ Enter and `flag[`i`]` $=$ `true` after performing the action. The invariant holds after performing the action.

Thread i makes a transition from "Enter" to "Spin": This occurs at line `P08` which leaves `flag` unchanged. By the assumption that I_1 held before performing the action (think of it as the "induction hypothesis"), `flag[`i`]` must have been `true` before performing the action. Thus $mode(i) =$ Spin and `flag[`i`]` $=$ `true` after performing the action. The invariant holds after performing the action.

Thread i makes a transition from "Spin" to "Crit": By an argument similar to the one above, `flag[`i`]` must be true before and after performing the action. Thus, the invariant continues to hold.

Thread i makes a transition from "Crit" to "Idle": This occurs at line `P13` which sets `flag` to `false`. Thus $mode(i) =$ Idle and `flag[`i`]` $=$ `false` after performing the action. The invariant holds after performing the action.

It's easy to see that I_1 holds in the initial state. However, $I_1 \Rightarrow Q$ is not a logical identity – in particular, I_1 allows both threads to be in mode "Crit" if `flag[0]` and `flag[1]` are both true. Our invariant needs to be a bit more specific.

Let's try $I_1 \wedge Q$. Clearly, any state that satisfies $I_1 \wedge Q$ satisfies Q. We've already shown that the initial state satisfies I_1 and that it satisfies Q, thus, it satisfies $I_1 \wedge Q$. If we could show that $I_1 \wedge Q$ is an invariant, we'd be done. Unfortunately, it is not. We can try repeating the arguments that we gave for showing that I_1 is an invariant. Three of the four cases work. We hit a problem when we consider:

Thread i makes a transition from "Spin" to "Crit": Consider the case that thread $i = 1$ and $mode(0) =$ Crit; in other words, thread 1 is spinning, and thread 0 is in its critical section. Looking at the "spin" statement:

```
P09:   while(flag[otherId] && (victim == myId)); % spin
```

If thread 1 is in its critical section, then `flag[0]` will be set, and thread 1 will only execute its spin-loop if `victim != 1`. Consider the state

$$s \quad = \quad \neg\{[\text{Crit, Spin}],\ [\texttt{true}, \texttt{true}],\ 0\}$$

The state s satisfies $I_1 \wedge Q$. However, if we perform one step of thread 1, we reach the state

$$s' \quad = \quad \neg\{[\text{Crit, Crit}],\ [\texttt{true}, \texttt{true}],\ 0\}$$

and mutual exclusion is violated.

Again, we ask the question: "Why do we believe that state s is unreachable?" We might guess that we need to show that when a thread i is spinning, that `victim == `i, but that's not quite right – if a thread makes an uncontested request for the lock, it will reach the "Spin" mode with `victim` set to itself. We need to be a bit more specific. After we think about it a bit, we realize that the issue is that if thread i is spinning *and the other thread is in its critical region*, then the spinning thread must be the victim. As a logic formula we write:

$$(mode(i) = \text{Spin}) \wedge (mode(1-i) = \text{Crit}) \quad \Rightarrow \quad \texttt{victim} = i$$

We include this in our invariant and get:

$$\begin{array}{lll} I_2 & \equiv & \forall i \in \{0,1\}.\ \texttt{flag[}i\texttt{]} = (mode(i) \in \{\text{Enter, Spin, Crit}\}) \\ & \wedge & \forall i \in \{0,1\}.\ ((mode(i) = \text{Spin}) \wedge (mode(1-i) = \text{Crit})) \Rightarrow (\texttt{victim} = i) \\ & \wedge & \texttt{victim} \in \{0,1\} \\ & \wedge & \neg((mode(0) = \text{Crit}) \wedge (mode(1) = \text{Crit})) \end{array}$$

I added the clause about `victim` because I'll need it in the proof – it's pretty obvious that it holds. Again, we prove the invariant by considering each action of each thread:

Thread j makes a transition from "Idle" to "Enter": Pretty much the same argument as for I_1. You can check your understanding of the approach by writing out why if I_2 holds before this action, it must hold afterwards.

Thread j makes a transition from "Enter" to "Spin": This occurs at line `P08` which leaves `flag` unchanged, and sets `victim` $= i$. The first clause of I_2 holds by the same argument as used for I_1. The second clause holds for $i = j$ because the action sets `victim` to j. The second clause holds for $i \neq j$ because the action sets $mode(j)$ to "Spin"; thus,

$$mode(1-i) \quad = \quad mode(j) \quad = \quad \text{Spin} \quad \neq \quad \text{Crit}$$

The third clause holds because `victim` is set to j and j is either 0 or 1. The fourth clause holds because the action sets $mode(j)$ to "Spin" which ensures that there is a thread whose mode is not "Crit".

Thread j makes a transition from "Spin" to "Crit": The first clause of I_2 holds by the same argument as used for I_1.

We now show that the second clause of I_2 hold after performing this action:

$$\forall i \in \{0,1\}.\ ((mode(i) = \text{Spin}) \wedge (mode(1-i) = \text{Crit})) \Rightarrow (\texttt{victim} = i)$$

After this action is performed, this thread will be in mode "Crit"; so the clause will hold for $i = j$ (because the left side of the implication will be false).

If the other thread is spinning, then the left side of the implication will be satisfied for $i = 1 - j$ after this action has been performed. Thus, we need to show that `victim` $= 1 - j$ will hold as well. By the assumption that the other thread is spinning, `flag`$[1-j]$ holds. Because thread j was enabled to exit its spin-loop, we conclude that `victim` $\neq j$. From the third clause of the invariant, we get that `victim` must be 0 or 1; thus `victim` $= 1 - j$. This establishes the right side of the implication (of the second clause of the invariant) for $i = j - 1$ in the case that the other thread is spinning. If the other thread is not spinning, then the second clause is satisfied for $i = j - 1$ because the left side of the implication is false.

The third clause holds because the value `victim` is not modified.

Now we get to the fourth clause of the invariant, mutual exclusion. This is the case that we needed to repair from our previous attempt. As noted above, if thread j can exit the while-loop at line `P09`, then either `flag[1-j]` $=$ `false`, or `victim` $\neq j$. Now consider the second clause of the invariant with $i = j$

$$((mode(j) = \text{Spin}) \wedge (mode(1-j) = \text{Crit})) \Rightarrow (\texttt{victim} = j)$$

We have shown that prior to performing this action, $mode(j) = \text{Spin}$ and `victim` $= j$. Therefore, $mode(1 - j) \neq \text{Crit}$ before performing the action. Because this action does not change $mode(1-j)$ (the *other* thread), $mode(1-j) \neq \text{Crit}$ holds after performing this action. Thus, the fourth clause of I_2 hold after performing this action.

We've shown that all clauses of I_2 hold after performing this action; therefore, I_2 holds after performing this action.

Thread i makes a transition from "Crit" to "Idle": Pretty much the same argument as for I_1. You can check your understanding of the approach by writing out why if I_2 holds before this action, it must hold afterwards.

We've shown that I_2 is an invariant. I_2 hold in the initial state and $I_2 \Rightarrow Q$. From this, we conclude that Q holds in all reachable states of the program.
HOORAY!!! We've shown that Peterson's algorithm guarantees mutual exclusion.

6 Generalizing the recipe

When trying to prove that some property holds in all states of a concurrent program, here are some steps that are often helpful.

1. Identify "modes" of the program. In HW4, Q4, I provided these and called them states. More generally, it's good to look for key transitions (e.g. entering or leaving a critical region), or statements that change the values of shared variables that are used to coordinate the actions of the threads. In the example above, this changed the program from having 23 PC values per thread to 4 modes (again, per thread), a big simplification.

2. Consider a few executions of the program to get an intuitive understanding of how it works.

3. Try writing an invariant that corresponds to your intuitive understanding of how the program works.

4. Verify the invariant:

   (a) Does it hold in the initial state?

   (b) Can you show that for each action of each thread, the invariant holding before the action implies that the invariant holds after the action? Note that the proposed invariant is your idea of what relations have to hold between the shared variables to make the program work, to show that it can't move to a "bad" state.

   (c) Does your invariant imply the property that you're trying to show?

5. Often, your first try for an invariant won't quite work. Typically, you get a counterexample. The counter-example is either a state in which the invariant holds but performing an action by some thread leads to a state that violates the invariant, or its a state in which the invariant holds, but the property that you're trying to show (e.g. mutual exclusion) is violated. Often, by thinking about that counterexample state, you can recognize why the program can't actually get to states like that. Then, you can add another clause or two to your proposed invariant and try again.

6. Of course, if you have a program that has a bug in it, you won't be able to find an invariant that proves the program correct. In this case, the counter-examples can lead you to finding an example execution that shows how the program can fail.

7 Review

Solve HW4.Q4.