What is the inductive invariant of the simple concurrent program?

Here is a simple concurrent program from the article Teaching Concurrency by Leslie Lamport.

Consider N processes numbered from 0 through $N-1$ in which each process i executes

```
x[i] := 1
y[i] := x[(i - 1) % N]
```

and stops, where each `x[i]` initially equals 0. (The reads and writes of each `x[i]` are assumed to be atomic.)

This algorithm satisfies the following property: after every process has stopped, `y[i]` equals 1 for at least one process i. It is easy to verify: The last process i to write `y[i]` must set it to 1.

Then, Lamport remarks that

> But that process doesn't set `y[i]` to 1 because it was the last process to write y.
>
> The algorithm satisfies this property because it maintains an **inductive invariant**. Do you know what that invariant is? If not, then you do not completely understand why the algorithm satisfies this property.

Therefore,

> What is the inductive invariant of the concurrent program?

algorithm concurrency invariants correctness

edited Jul 29 '14 at 15:00

asked Jul 28 '14 at 6:44
hengxin
713 2 7 25

2 Answers

The `x` s model the following behavior: `x[i]` is `1` if and only if the process `i` has already run. Naturally, after all processes have completed, all `x` s are thus set to `1` .

The `y` s are a bit trickier: `y[i]` is set if `x[i-1]` was set, that is, `y[i]` is `1` if and only if *the predecessor* of `i` had already run when `i` was doing its write to `y` .

The program invariant is: *If a process has set* `y[i]` *, it must already have set* `x[i]` *to* `1` . This is true regardless whether `y[i]` is set to `0` or `1` .

Proving this invariant is quite easy: In the beginning, none of the `y` s is set, so it holds trivially. During program execution, each write to `y[i]` is sequenced after a write to `x[i]` . Therefore the invariant also holds for every step of the program afterwards.

The further reasoning goes like this: The last process to finish sets `y[i]` to `1` because, by definition of being the last process, its predecessor must have already finished execution at that point (ie. its `y` value is already set). Which means, because of the invariant, its `x` value (which determines the last process' `y` value) has to be `1` .

Another way to look at it: You cannot find an execution order in which all `y` s are set to `0` . Doing so would require all processes to execute before their predecessor. However, since our processes are arranged in a ring (that is, if I follow the predecessor chain I will eventually end up at my starting point again), this leads to the contradiction that at least one process must have finished executing before it wrote to `y` .

edited Jul 30 '14 at 7:24

answered Jul 29 '14 at 16:20
ComicSansMS
25.3k 3 71 105

Your two arguments are of course reasonable. However I doubt whether it is the **inductive invariant** in Lamport's mind (surely we will never know this). Being an inductive invariant `I` , I would like to see an argument by *mathematical induction*: Initially, `I` holds trivially; Suppose `I` holds at some time point, it still holds after some little thing happens; such that when all processes finish, it naturally implies that the property to prove is satisfied. In addition, an inductive invariant is more likely to be global (not local). What is your opinion? – hengxin Jul 30 '14 at 1:47

@hengxin I think the invariant I gave is indeed inductive. I tried to edit the answer to make this more clear. The program itself is rather trivial, so the invariant is really not too spectacular, but it suffices to give a proof of correctness. In this case however I personally find the proof by contradiction given in the last paragraph (which does not rely on the invariant) more intuitive. For an automated proof (which is what Lamport ultimately is aiming at in this paper) however, the invariant approach is usually the more effective one. – ComicSansMS Jul 30 '14 at 7:27

TL;DR

An inductive invariant of this program, in TLA+ syntax, is:

```
/\ \A i \in 0..N-1 : (pc[i] \in {"s2", "Done"} => x[i] = 1)
/\ (\A i \in 0..N-1 : pc[i] = "Done") => \E i \in 0..N-1 : y[i] = 1
```

What is an inductive invariant?

An inductive invariant is an invariant that satisfies the following two conditions:

```
Init => Inv
Inv /\ Next => Inv'
```

where:

- `Inv` is the inductive invariant
- `Init` is the predicate that describes the initial state
- `Next` is the predicate that describes state transitions.

Why use inductive invariants?

Note that an inductive invariant is only about current state and next state. It makes no references to the execution history, it's not about the past behavior of the system.

In section 7.2.1 of Principles and Specifications of Concurrent Systems (generally known as The TLA+ Hyperbook), Lamport describes why he prefers using inductive invariants over behavioral proofs (i.e., those that make reference to execution history).

> Behavioral proofs can be made more formal, but I don't know any practical way to make them completely formal—that is, to write executable descriptions of real algorithms and formal behavioral proofs that they satisfy correctness properties. This is one reason why, in more than 35 years of writing concurrent algorithms, I have found behavioral reasoning to be unreliable for more complicated algorithms. I believe another reason to be that behavioral proofs are inherently more complex than state-based ones for sufficiently complex algorithms. This leads people to write less rigorous behavioral proofs for those algorithms—especially with no completely formal proofs to serve as guideposts.
>
> To avoid mistakes, we have to think in terms of states, not in terms of executions... Still, behavioral reasoning provides a different way of thinking about an algorithm, and thinking is always helpful. Behavioral reasoning is bad only if it is used instead of state-based reasoning rather than in addition to it.

Some preliminaries

The property we are interested in proving is (in TLA+ syntax):

```
(\A i \in 0..N-1 : pc[i] = "Done") => \E i \in 0..N-1 : y[i] = 1
```

Here I'm using the PlusCal convention of describing the control state of each process using a variable named "pc" (I think of it as "program counter").

This property is an invariant, but it's not an inductive invariant, because it doesn't satisfy the conditions above.

You can use an inductive invariant to prove a property by writing a proof that look like this:

```
1. Init => Inv
2. Inv /\ Next => Inv'
3. Inv => DesiredProperty
```

To come up with an inductive invariant, we need to give labels to each step of the algorithm, let's call them "s1", "s2", and "Done", where "Done" is the terminal state for each process.

```
s1: x[self] := 1;
s2: y[self] := x[(self-1) % N]
```

Coming up with an inductive invariant

Consider the state of the program when it is in the penultimate (second-to-last) state of an execution.

In the last execution state, `pc[i]="Done"` for all values of i. In the penultimate state, `pc[i]="Done"` for all values of i except one, let's call it j, where `pc[j]="s2"` .

If a process i is in the "Done" state, then it must be true that `x[i]=1` , since the process must have executed statement "s1". Similarly, the process j that is in the state "s2" must also have executed statement "s1", so it must be true that `x[j]=1` .

We can express this as an invariant, which happens to be an inductive invariant.

```
\A i \in 0..N-1 : (pc[i] \in {"s2", "Done"} => x[i] = 1)
```

PlusCal model

To prove that our invariant is an inductive invariant, we need a proper model that has an `Init` state predicate and a `Next` state predicate.

We can start by describing the algorithm in PlusCal. It's a very simple algorithm, so I'll call it "Simple".

```
--algorithm Simple

variables
    x = [i \in 0..N-1 |->0];
    y = [i \in 0..N-1 |->0];

process Proc \in 0..N-1
begin
    s1: x[self] := 1;
    s2: y[self] := x[(self-1) % N]
end process

end algorithm
```

Translating to TLA+

We can translate the PlusCal model into TLA+. Here's what it looks like when we translate PlusCal into TLA+ (I've left out the termination condition, because we don't need it here).

```
------------------------------ MODULE Simple ------------------------------

EXTENDS Naturals

CONSTANTS N

VARIABLES x, y, pc

vars == << x, y, pc >>

ProcSet == (0..N-1)

Init == (* Global variables *)
        /\ x = [i \in 0..N-1 |->0]
        /\ y = [i \in 0..N-1 |->0]
        /\ pc = [self \in ProcSet |-> "s1"]

s1(self) == /\ pc[self] = "s1"
            /\ x' = [x EXCEPT ![self] = 1]
            /\ pc' = [pc EXCEPT ![self] = "s2"]
            /\ y' = y

s2(self) == /\ pc[self] = "s2"
            /\ y' = [y EXCEPT ![self] = x[(self-1) % N]]
            /\ pc' = [pc EXCEPT ![self] = "Done"]
            /\ x' = x

Proc(self) == s1(self) \/ s2(self)

Next == (\E self \in 0..N-1: Proc(self))
           \/ (* Disjunct to prevent deadlock on termination *)
              ((\A self \in ProcSet: pc[self] = "Done") /\ UNCHANGED vars)

Spec == Init /\ [][Next]_vars
=============================================================================
```

Note how it defines the `Init` and `Next` state predicates.

The inductive invariant in TLA+

We can now specify the inductive invariant we want to prove. We also want our inductive invariant to imply the property we're interested in proving, so we add it as a conjunction.

```
Inv == /\ \A i \in 0..N-1 : (pc[i] \in {"s2", "Done"} => x[i] = 1)
       /\ (\A i \in 0..N-1 : pc[i] = "Done") => \E i \in 0..N-1 : y[i] = 1
```

Informal handwaving "proof"

1. **`Init => Inv`**

It should be obvious why this is true, since the antecedents in `Inv` are all false if `Init` is true.

2. **`Inv /\ Next => Inv'`**

The first conjunct of Inv'
`(\A i \in 0..N-1 : (pc[i] \in {"s2", "Done"} => x[i] = 1))'`

The interesting case is the one where `pc[i]="s1"` and `pc'[i]="s2"` for some i. By the definition of `s1`, it should be clear why this is true.

The second conjunct of Inv'
`((\A i \in 0..N-1 : pc[i] = "Done") => \E i \in 0..N-1 : y[i] = 1)'`

The interesting case is the one we discussed earlier, where `pc[i]="Done"` for all values of i except one, j, where `pc[j]="s2"`.

By the first conjunct of Inv, we know that `x[i]=1` for all values of i.

By `s2` , `y'[j]=1` .

3. **`Inv => DesiredProperty`**

Here, our desired property is

```
(\A i \in 0..N-1 : pc[i] = "Done") => \E i \in 0..N-1 : y[i] = 1
```

Note that we've just anded the property that we are interested in to the invariant, so this is trivial to prove.

Formal proof with TLAPS

You can use the TLA+ Proof System (TLAPS) to write a formal proof that can be checked mechanically to determine if it is correct.

Here's a proof that I wrote and verified using TLAPS that uses an inductive invariant to prove the desired property. (Note: this is the first proof I've ever written using TLAPS, so keep in mind this has been written by a novice).

```
AtLeastOneYWhenDone == (\A i \in 0..N-1 : pc[i] = "Done") => \E i \in 0..N-1 : y[i]
= 1

TypeOK == /\ x \in [0..N-1 -> {0,1}]
          /\ y \in [0..N-1 -> {0,1}]
          /\ pc \in [ProcSet -> {"s1", "s2", "Done"}]

Inv == /\ TypeOK
       /\ \A i \in 0..N-1 : (pc[i] \in {"s2", "Done"} => x[i] = 1)
       /\ AtLeastOneYWhenDone

ASSUME NIsInNat == N \in Nat \ {0}

\* TLAPS doesn't know this property of modulus operator
AXIOM ModInRange == \A i \in 0..N-1: (i-1) % N \in 0..N-1


THEOREM Spec=>[]AtLeastOneYWhenDone
<1> USE DEF ProcSet, Inv
<1>1. Init => Inv
    BY NIsInNat DEF Init, Inv, TypeOK, AtLeastOneYWhenDone
<1>2. Inv /\ [Next]_vars => Inv'
  <2> SUFFICES ASSUME Inv,
                      [Next]_vars
               PROVE  Inv'
    OBVIOUS
  <2>1. CASE Next
    <3>1. CASE \E self \in 0..N-1: Proc(self)
      <4> SUFFICES ASSUME NEW self \in 0..N-1,
                          Proc(self)
                   PROVE  Inv'
        BY <3>1
      <4>1. CASE s1(self)
        BY <4>1, NIsInNat DEF s1, TypeOK, AtLeastOneYWhenDone
      <4>2. CASE s2(self)
        BY <4>2, NIsInNat, ModInRange DEF s2, TypeOK, AtLeastOneYWhenDone
      <4>3. QED
        BY <3>1, <4>1, <4>2 DEF Proc
    <3>2. CASE (\A self \in ProcSet: pc[self] = "Done") /\ UNCHANGED vars
        BY <3>2 DEF TypeOK, vars, AtLeastOneYWhenDone
    <3>3. QED
      BY <2>1, <3>1, <3>2 DEF Next
  <2>2. CASE UNCHANGED vars
    BY <2>2 DEF TypeOK, vars, AtLeastOneYWhenDone
  <2>3. QED
    BY <2>1, <2>2
<1>3. Inv => AtLeastOneYWhenDone
    OBVIOUS
<1>4. QED
    BY <1>1, <1>2, <1>3, PTL DEF Spec
```

Note that in a proof using TLAPS, you need to have a type checking invariant (it's called `TypeOK` above), and you also need to handle "stuttering states" where none of the variables change, which is why we use `[Next]_vars` instead of `Next` .

Here's a gist with the complete model and proof.

edited 5 hours ago

answered 5 hours ago

Lorin Hochstein
27.4k 17 79 117

Answer Your Question