

Liveness Properties

Jens Chr. Godskesen
IT University of Copenhagen

Overview

- The Readers/Writers Problem
- Liveness and Strong Liveness Properties
- Priorities to Writers
- Fairness and Individual Fairness
- Exercises

Readers and Writers

The **readers-writers problem** is about access to a database. Several processes may *read* from the database simultaneously, but *writing* has to be done under mutual exclusion.

The problem is close to the mutual exclusion problem in that the readers and writers are in a race condition, but they are divided into two categories:

Readers Processes not excluding one another

Writers Processes excluding every other process

Modelling Readers/Writers

Two readers and writers may be modelled by:

```
agent Reader(acqR,relR,read) =
    'acqR.read.'relR.Reader<acqR,relR,read>
agent Writer(acqW,relW,write) =
    'acqW.write.'relW.Writer<acqW,relW,write>

agent Lock0(acqW,relW,acqR,relR) =
    acqW.relW.Lock0<acqW,relW,acqR,relR> + acqR.Lock1<acqW,relW,acqR,relR>
agent Lock1(acqW,relW,acqR,relR) =
    acqR.Lock2<acqW,relW,acqR,relR> + relR.Lock0<acqW,relW,acqR,relR>
agent Lock2(acqW,relW,acqR,relR) =
    relR.Lock1<acqW,relW,acqR,relR>

agent Sys(r1,r2,w1,w2) =
    (^ aR,rR,aW,rW) ( Reader<aR,rR,r1> | Reader<aR,rR,r2>
                    | Writer<aW,rW,w1> | Writer<aW,rW,w2>
                    | Lock0<aW,rW,aR,rR>)
```

Checking Safety Properties

```
MWB>prove Sys<r1,r2,w1,w2>  
      nu X.( ( [r1]FF | [w1]FF )  
             & ( [t]X & [r1]X & [r2]X & [w1]X & [w2]X ) )  
Model Prover says: YES!
```

```
MWB>prove Sys<r1,r2,w1,w2>  
      nu X.( ( [w1]FF | [w2]FF )  
             & ( [t]X & [r1]X & [r2]X & [w1]X & [w2]X ) )  
Model Prover says: YES!
```

```
MWB>prove Sys<r1,r2,w1,w2>  
      nu X.( ( [r1]FF | [r2]FF )  
             & ( [t]X & [r1]X & [r2]X & [w1]X & [w2]X ) )  
Model Prover says: NO.
```

Liveness Properties

In contrast to safety properties that are about absence of something bad, **liveness properties** are statements about the presence of something.

Liveness properties are properties stating that “something good will eventually happens” .

A liveness property is *eventually* true, i.e. it holds true in some state in the execution tree.

A liveness property must ensure *progress*.

Ensuring Progress

The safety property

$$\nu X.(\phi \wedge \bigwedge_{a \in \mathcal{A}} [a]X)$$

does not ensure progress.

$\bigwedge_{a \in \mathcal{A}} [a]X$ only ensures that *if* there exists a transition then for all transitions the property holds recursively.

A safety property may be satisfied by the inactive process, e.g.

```
MWB>prove 0 nu X.( ( [r1]FF | [w1]FF )  
                & ( [t]X & [r1]X & [r2]X & [w1]X & [w2]X ) )
```

Model Prover says: YES!

Ensuring Progress, continued

A liveness property may be specified as a solution to the equation

$$X = \phi \vee \bigvee_{a \in \mathcal{A}} \langle a \rangle X$$

i.e. either ϕ holds in the current state, or for some transition leaving the current state the property holds recursively.

Notice how the subformula $\langle a \rangle X$ enforces progress in case ϕ is not satisfied yet.

The Liveness Formula

The maximal solution to $X = \phi \vee \bigvee_{a \in \mathcal{A}} \langle a \rangle X$ is the limit of

$$tt \supseteq \phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle tt \supseteq \phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle (\phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle tt) \supseteq \dots$$

Approximation k is the set of processes where each process

- either can perform a sequence of transitions of length at least $k - 1$, or
- it can perform a sequence of transitions of length less than $k - 1$ to a state in which ϕ is satisfied.

Example

Let $\phi = \langle b \rangle tt$ and define

$$P = b.0 \quad \text{and} \quad Q = a.Q$$

Substituting ϕ into $\phi' = \nu X.(\phi \vee \langle a \rangle X \vee \langle b \rangle X)$ it follows that

- $P \models \phi'$ because $P \xrightarrow{b} 0$, and that
- $Q \models \phi'$ because it can do an infinite sequence of a 's

However, since Q never eventually performs a b then clearly the maximal solution to $X = \phi \vee \bigvee_{a \in \mathcal{A}} \langle a \rangle X$ is not what we want.

Minimal Fixed Points

The smallest solution to $X = \phi \vee \bigvee_{a \in \mathcal{A}} \langle a \rangle X$ is the right choice, i.e. liveness should be defined as a *minimal fixed point*.

We extend HML by a *minimal fixed point operator*

$$\phi ::= \dots \mid \mu X. \phi$$

and define liveness properties by

$$\mu X. (\phi \vee \bigvee_{a \in \mathcal{A}} \langle a \rangle X)$$

Intuition, Minimal Fixed Point

The minimal fixed point of $X = \phi \vee \bigvee_{a \in \mathcal{A}} \langle a \rangle X$ is the limit of the increasing sequence

$$ff \subseteq \phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle ff \subseteq \phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle (\phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle ff) \subseteq \dots$$

Since $\langle a \rangle ff$ is the same as ff we get instead

$$ff \subseteq \phi \subseteq \phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle \phi \subseteq \phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle (\phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle \phi) \subseteq \dots$$

i.e. any processes in approximation $k+1$ satisfy ϕ after a sequence of transitions of length at most $k - 1$.

Examples

- Let $Q = a.Q$ and let $\phi = \mu X.(\langle b \rangle tt \vee \langle a \rangle X \vee \langle b \rangle X)$, then $Q \not\models \phi$ because it cannot ever do a b .
- In the readers/writers, will the writers eventually be allowed to write? In MWB we get

```
MWB>prove Sys<r1,r2,w1,w2>  
      mu X.( ( <w1>TT | <w2>TT )  
            | ( ( <t>X ) | <r1>X | <r2>X | <w1>X | <w2>X ) )
```

Model Prover says: YES!
(106 inferences)

Will the property still hold if we replace $\langle w1 \rangle tt \vee \langle w2 \rangle tt$ by $\langle w1 \rangle tt$? What about replacing by $\langle r1 \rangle tt$?

Readers/Writers Implementation

```
public class ReadersWriters {  
  
    static int numReaders = 2;  
    static int numWriters = 2;  
  
    public static void main(String[] argv) {  
  
        Database d = new Database();  
  
        try {  
            for (int i =0; i<numReaders; ++i)  
                new Reader(d,i).start();  
            for (int i =0; i<numWriters; ++i)  
                new Writer(d,i+numReaders).start();  
        } catch (Exception e) {System.out.println("Exception: "+e);}  
    }  
}
```

Readers

```
public class Reader extends Thread {  
  
    Database d;  
    int name;  
  
    Reader(Database d, int n) { this.d = d; name = n; }  
  
    public void run() {  
        try {  
            while (true) {  
                d.acqR(name);  
                Thread.sleep(3500);  
                if (Math.random()<0.5) Thread.sleep(2500);  
                System.out.println("Reader " + name + " reads " + d.read());  
                d.relR(name);  
            }  
        } catch (InterruptedException e){}  
    }  
}
```

Writers

```
public class Writer extends Thread {  
  
    Database d;  
    int name;  
  
    Writer(Database d, int n) { this.d = d; name = n; }  
  
    public void run() {  
        try {  
            while (true) {  
                d.acqW(name);  
                Thread.sleep(3500);  
                if (Math.random()<0.5) Thread.sleep(2500);  
                d.write(name);  
                System.out.println("Writer " + name + " writes");  
                d.relW(name);  
            }  
        } catch (InterruptedException e){}  
    }  
}
```

The Database

```
public class Database {  
  
    private int content, readLocks;  
    private boolean writeLocked;  
  
    public Database() {  
        content = readLocks = 0;  
        writeLocked = false;  
    }  
  
    synchronized public void write(int i) {  
        content = i;  
    }  
  
    synchronized public int read() {  
        return content;  
    }  
}
```

The Database, continued

```
synchronized public void acqR(int name) throws InterruptedException {  
    while (writeLocked) wait();  
    System.out.println("Reader " + name + " got the lock");  
    readLocks++;  
}
```

```
synchronized public void relR(int name) {  
    System.out.println("Reader " + name + " releases the lock");  
    readLocks--;  
    if (readLocks == 0) notify();  
}
```

The Database, continued

```
synchronized public void acqW(int name) throws InterruptedException {
    while (readLocks > 0 || writeLocked) wait();
    System.out.println("Writer " + name + " got the lock");
    writeLocked = true;
}

synchronized public void relW(int name) {
    System.out.println("Writer " + name + " releases the lock");
    writeLocked = false;
    notifyAll();
}
}
```

Understanding and Running the Program

Exercise Explain the notification principles of the database.

Exercise Try to run the program `ReadersWriters.java`. Explain the result.

Strengthening the Liveness Property

The starvation of writers is not in conflict with the liveness property $\mu X.(\phi \vee \bigvee_{a \in \mathcal{A}} \langle a \rangle X)$ where $\phi = \langle w1 \rangle tt \vee \langle w2 \rangle tt$.

It only states that there is a reachable state (along some path) in the computation tree satisfying ϕ .

For the program to be *fair*, in the sense that in any run the writers will eventually be writing to the database, it must satisfy the stronger progress property:

any path eventually satisfies $\langle w1 \rangle tt \vee \langle w2 \rangle tt$

Strong Liveness

The *Strong Liveness* property is defined by

$$\mu X.(\phi \vee (\bigvee_{a \in \mathcal{A}} \langle a \rangle tt \wedge \bigwedge_{a \in \mathcal{A}} [a]X))$$

Intuitively it means that

- either ϕ holds in the current state, or
- there is progress ($\bigvee_{a \in \mathcal{A}} \langle a \rangle tt$), and no matter how progress is ensured then the formula holds recursively ($\bigwedge_{a \in \mathcal{A}} [a]X$).

Why a Minimal Fixed Point?

Why a minimal solution to $X = \phi \vee (\bigvee_{a \in \mathcal{A}} \langle a \rangle tt \wedge \bigwedge_{a \in \mathcal{A}} [a]X)$?

The maximal solution is the limit to

$$tt \supseteq \phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle tt \supseteq \phi \cup \left(\bigcup_{a \in \mathcal{A}} \langle a \rangle tt \cap \bigcap_{a \in \mathcal{A}} [a] (\phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle tt) \right) \supseteq \dots$$

whereas the minimal solution is the limit to

$$\begin{aligned} ff &\subseteq \phi \subseteq \phi \cup \left(\bigcup_{a \in \mathcal{A}} \langle a \rangle tt \cap \bigcap_{a \in \mathcal{A}} [a] \phi \right) \\ &\subseteq \phi \cup \left(\bigcup_{a \in \mathcal{A}} \langle a \rangle tt \cap \bigcap_{a \in \mathcal{A}} [a] (\phi \cup \left(\bigcup_{a \in \mathcal{A}} \langle a \rangle tt \cap \bigcap_{a \in \mathcal{A}} [a] \phi \right)) \right) \subseteq \dots \end{aligned}$$

If $P = a.P$ and $\phi = \langle b \rangle tt$ then P satisfy the maximal but not the minimal solution.

Example

Make a drawing of the labelled transition system for the model of the readers and writers.

Argue why the model of the readers/writers doesn't satisfy

$$\mu X.(\phi \vee (\bigvee_{a \in \mathcal{A}} \langle a \rangle tt \wedge \bigwedge_{a \in \mathcal{A}} [a]X))$$

if we instantiate ϕ by $\langle w1 \rangle tt \vee \langle w2 \rangle tt$.

What if we instantiate ϕ by $\langle r1 \rangle tt \vee \langle r2 \rangle tt$?

Giving Priority to Writers

It may be realistic to give *priority* to writers since often the most recent updates are wanted.

We let the writers make a request before acquiring the lock, hence a writer becomes

```
agent Writer(reqW,acqW,relW,write) =  
    'reqW.'acqW.write.'relW.Writer<reqW,acqW,relW,write>
```

The lock is modified to prohibit readers getting the lock if writers are waiting.

The Modified Lock Model

```
agent Lock00(reqW,acqW,relW,acqR,relR) =
  reqW.Lock01<reqW,acqW,relW,acqR,relR> + acqR.Lock10<reqW,acqW,relW,acqR,relR>
agent Lock01(reqW,acqW,relW,acqR,relR) =
  reqW.Lock02<reqW,acqW,relW,acqR,relR> + acqW.relW.Lock00<reqW,acqW,relW,acqR,relR>
agent Lock02(reqW,acqW,relW,acqR,relR) =
  acqW.relW.Lock01<reqW,acqW,relW,acqR,relR>
agent Lock10(reqW,acqW,relW,acqR,relR) =
  reqW.Lock11<reqW,acqW,relW,acqR,relR> + acqR.Lock20<reqW,acqW,relW,acqR,relR>
  + relR.Lock00<reqW,acqW,relW,acqR,relR>
agent Lock11(reqW,acqW,relW,acqR,relR) =
  reqW.Lock12<reqW,acqW,relW,acqR,relR> + relR.Lock01<reqW,acqW,relW,acqR,relR>
agent Lock12(reqW,acqW,relW,acqR,relR) =
  relR.Lock02<reqW,acqW,relW,acqR,relR>
agent Lock20(reqW,acqW,relW,acqR,relR) =
  reqW.Lock21<reqW,acqW,relW,acqR,relR> + relR.Lock10<reqW,acqW,relW,acqR,relR>
agent Lock21(reqW,acqW,relW,acqR,relR) =
  reqW.Lock22<reqW,acqW,relW,acqR,relR> + relR.Lock11<reqW,acqW,relW,acqR,relR>
agent Lock22(reqW,acqW,relW,acqR,relR) =
  relR.Lock12<reqW,acqW,relW,acqR,relR>
```

The Modified DB Implementation

```
private int ... writeWaiting = 0;

synchronized public void acqR(int name) throws InterruptedException {
    while (writeLocked || writeWaiting > 0) wait(); // A
    System.out.println("Reader " + name + " got the lock");
    readLocks++;
}

synchronized public void relR(int name) {
    System.out.println("Reader " + name + " releases the lock");
    readLocks--;
    if (readLocks == 0) notifyAll(); // B
}
```

The Modified DB Implementation, continued

```
synchronized public void acqW(int name) throws InterruptedException {
    writeWaiting++;    // the request                // C
    while (readLocks > 0 || writeLocked) wait();
    writeWaiting--;    // D
    writeLocked = true;
    System.out.println("Writer " + name + " got the lock");
}

synchronized public void relW(int name) {
    System.out.println("Writer " + name + " releases the lock");
    writeLocked = false;
    notifyAll();
}
```

Properties

```
agent Sys(r1,r2,w1,w2) =  
  ( ^ aR,rR,qW,aW,rW) (Reader<aR,rR,r1> | Reader<aR,rR,r2>  
    | Writer<qW,aW,rW,w1> | Writer<qW,aW,rW,w2>  
    | Lock00<qW,aW,rW,aR,rR> )
```

satisfies the safety properties: no two writers must access the database simultaneously, and neither may a writer and a reader.

The strong liveness property with ϕ replaced by $\langle w1 \rangle tt \vee \langle w2 \rangle tt$ is *not* satisfied

There exists paths where no writer ever issue a request for writing to the database, corresponding to writers not ever being allowed to enter the monitor.

Ensuring Fairness

We continue under the assumption that all readers and writers are provided fair access to the database monitor.

We establish a **fair** solution to the readers/writers problem, i.e. a solution where both readers and writers eventually are granted access to the database.

The means is a boolean variable which value tells if it is a readers or a writers turn

The variable introduces asymmetry into the implementation such that readers cannot always be blocked because of waiting writers.

The Fair Database

```
private boolean ... readersTurn = true;

synchronized public void acqR(int name) throws InterruptedException {
    while (writeLocked || (writeWaiting > 0 && !readersTurn)) wait(); // A
    System.out.println("Reader " + name + " got the lock");
    readLocks++;
}

synchronized public void relR(int name) {
    System.out.println("Reader " + name + " releases the lock");
    readLocks--;
    readersTurn = false; // B
    if (readLocks == 0) notifyAll();
}
```

The Fair Database, continued

```
synchronized public void acqW(int name) throws InterruptedException {
    writeWaiting++;
    while (readLocks > 0 || writeLocked || readersTurn) wait();      // C
    writeWaiting--;
    writeLocked = true;
    System.out.println("Writer " + name + " got the lock");
}
```

```
synchronized public void relW(int name) {
    System.out.println("Writer " + name + " releases the lock");
    writeLocked = false;
    readersTurn = true;                                             // D
    notifyAll();
}
```

Why a fair database?

A writer has the lock and writers are waiting: Upon releasing the lock the turn is given to readers. Hence no writer will be allowed to obtain the lock next and since the `acqR` condition synchronization is false a reader is allowed to enter the monitor.

Several readers have the lock and writers are waiting: When a reader releases the lock it gives the turn to writers. Hence no readers will be allowed to acquire the lock. Eventually all readers have released the lock and a writer obtains the lock.

Exercise Try to run the program.

Individual Fairness

The previous database does not ensure **individual fairness** in the sense that one particular reader (writer) is eventually given the opportunity to read (write).

The same reader (writer) may always be granted access to the database.

To ensure individual fairness we introduce a FIFO ordering on the requests to the database using a *ticketing principle*.

Implementing the Ticketing Principle

```
private int ... ticket = 0, next = 0;

synchronized public void acqR(int name) throws InterruptedException {
    int myticket = ticket++; // got a ticket
    while (myticket != next) wait();
    System.out.println("Reader " + name + " with ticket "
        + myticket + " got the lock");
    readLocks++;
    next++;
    notifyAll(); // make sure to wake up reader that may hold next ticket
}

synchronized public void relR(int name) {
    System.out.println("Reader " + name + " releases the lock");
    readLocks--;
    if (readLocks == 0) notifyAll();
}
```

Implementing the Ticketing Principle, continued

```
synchronized public void acqW(int name) throws InterruptedException {
    int myticket = ticket++;
    while (readLocks > 0 || myticket != next) wait();
    System.out.println("Writer " + name + " with ticket "
        + myticket + " got the lock");
}

synchronized public void relW(int name) {
    System.out.println("Writer " + name + " releases the lock");
    next++;
    notifyAll();
}
```

Why a FIFO implementation?

A writer has the lock: The owner of the next ticket can be served after `next` is incremented in the writers call to `relW`.

A reader has the lock: If a reader has the next ticket she may start reading immediately since `next` is incremented in `acqR`. If a writer has the next ticket then she must wait writing until all readers have released their lock.

Exercise Try to run the program.

Exercises

1. Go through the exercises we didn't make during the lecture
2. What would be reasonable liveness properties for the Dining Philosophers? State and check the properties using MWB.
3. Explain why $P = a.P$ satisfy $\nu X.(\phi \vee (\bigvee_{a \in \mathcal{A}} \langle a \rangle tt \wedge \bigwedge_{a \in \mathcal{A}} [a] X))$ when ϕ is $\langle b \rangle tt$ although P can do no b .
4. Continue with the second mandatory assignment