

# Liveness and Fairness

Version 1.1

Jens Chr. Godskesen  
IT University of Copenhagen

©2004, 2005 Jens Chr. Godskesen

March 8, 2005

## 1 Introduction

This note is devoted to *liveness* and *progress* properties. They are properties stating intuitively that “something good will eventually happens”. For instance, the absence of starvation for any of the dining philosophers in [1] is a liveness property and so is the possibility of being allowed to copy in the printer and scanner example also presented in [1].

We shall see how progress and liveness properties can be specified in the recursive extension of HML [1], however as it turns out, these kind of properties we must specify using *minimal fixed points* instead of as for safety properties using the maximal fixed points.

For illustration purposes, we shall use the classical example of *readers and writers* as our running example.

The examples of models to be run in MWB in this note are presented with emphasis on readability and may thus not compile due to extra line breaks.

## 2 Readers and Writers

The readers-writers problem is a well-known and thoroughly studied example in concurrency theory.

The problem is about access to a database (regarding the notion of a database in a broad sense), where several processes perfectly may read from the database simultaneously, but writing has to be done under mutual exclusion; otherwise consistency may be lost.

The readers-writers problem is in its nature very close to the mutual exclusion problems, like the producer-consumer problem for instance, in that the readers and writers

processes are in a race condition (for accessing their critical section), but the processes are in this case divided into two distinct categories:

**Readers** Processes not excluding one another

**Writers** Processes excluding every other process

## 2.1 Modeling Readers-Writers

We start making a model for the solution considering only two readers and two writers that are synchronized through a lock:

```
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> )
```

Notice the lock, it allows at most one writer to obtain the lock at any one time, but only if the lock is not possessed by a reader. Moreover, a reader may obtain the lock even though it is taken by other readers. The index  $i$  in `Lock $i$`  refers to the number of readers simultaneously having obtained the lock.

## 2.2 Checking Safety Properties

Next we check various safety properties using the general format for safety properties presented in [1]

$$\nu X.(\phi \wedge \bigwedge_{a \in \mathcal{A}} [a]X) \quad (1)$$

First it would be obvious to verify that a reader and writer cannot access the database at the same time. Next, we should check that neither can two writers access the database at any one time. Finally, we check that it indeed is possible for two readers to access the database simultaneously.

Using MWB we get:

```

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.

```

As it turns out, the two properties stating that writers must have exclusive access to the database are both satisfied, and fortunately the third property is invalid because it should indeed be possible for two readers to access the database concurrently.

### 3 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 bit more formally it means that a property will eventually become true in some state (including the current state).

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

Notice that a liveness property must ensure *progress* in that if the “good thing” does not hold in the current state then the property must enforce that transitions are successively taken until the “good thing” holds in a reachable state.

#### 3.1 Ensuring Progress

In some way liveness properties are the dual of safety properties. Recall the general format of a safety property defined by (1) where  $\phi$  is supposed to hold in every state and where the subformula  $\bigwedge_{a \in \mathcal{A}} [a]X$  only ensures that if there exists a transition then for all transitions the property holds recursively.

Hence in (1) there is no guarantee that some transition must be taken, so a safety property does not ensure progress.

This fact implies that a safety property may be satisfied by the inactive process. Take for instance the first of the safety property we devised for the readers and writers above, then running MWB we get

```
MWB>prove 0 nu X.( ( [r1]FF | [w1]FF )
                & ( [t]X & [r1]X & [r2]X & [w1]X & [w2]X ) )
Model Prover says: YES!
```

because clearly 0 satisfies  $[r1]FF \mid [w1]FF$ , since it can neither read nor write, and since 0 has no outgoing transitions the recursive parts is taken care of vacuously.

Don't be misled by the example above and do notice that the inactive process do not satisfy all safety properties!

**Exercise** Why will the inactive process not satisfy the safety property for absence of deadlock where  $\phi = \bigvee_{a \in \mathcal{A}} \langle a \rangle tt$  is substituted into (1)?

From the discussion above one may choose to specify a liveness property as a solution to the equation

$$X = \phi \vee \bigvee_{a \in \mathcal{A}} \langle a \rangle X \quad (2)$$

which intuitively means that either  $\phi$  must hold in the current state or for some action  $a$  there is an  $a$ -transition leaving the current state such that the property holds recursively. Notice in particular how the subformula  $\langle a \rangle X$  enforces progress in case  $\phi$  is not satisfied yet.

## 4 The Liveness Formula

In [1] we saw how a solution to a recursively defined equation may be defined by a maximal fixed point as a limit of a decreasing sequence starting with an over approximation (being the set of all processes) and iteratively refining approximations toward a limit.

In case of the equation (2) the sequence would be:

$$tt \supseteq \phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle tt \supseteq \phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle (\phi \cup \bigcup_{a \in \mathcal{A}} \langle a \rangle tt) \supseteq \dots \quad (3)$$

Hence the first approximation is all processes. The second approximation is all processes that either satisfy  $\phi$  in the current state or can do some transition. The third approximation is all processes that either satisfy  $\phi$  in the current state or can do some transition to a state where it either satisfy  $\phi$  or where it can do a transition.

Intuitively, for any approximation  $k$  it turns out that it defines 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  $\phi$  is satisfied.

As an example, let  $\phi$  be the formula  $\langle b \rangle tt$  and define two CCS processes

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

Then substituting  $\phi$  into (2) and taking the maximal solution as indicated by (3) it follows that both  $P$  and  $Q$  satisfy the property. However, since  $Q$  never eventually performs a  $b$  then clearly the maximal solution to (2) is not what we want.

#### 4.1 Taking the minimal fixed point

It turns out that the smallest solution is the right choice for liveness (and progress) properties, or in technical terms that the liveness property should be defined as a *minimal fixed point*. Hence we extend HML yet again this time by a *minimal fixed point operator*

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

and define liveness properties by

$$\mu X. (\phi \vee \bigvee_{a \in \mathcal{A}} \langle a \rangle X) \quad \text{Liveness} \quad (4)$$

assuming (as for the safety property)  $\mathcal{A}$  to be finite.

Formally a minimal fixed point is defined as the limit of an increasing sequence, for instance in case of the formula for liveness we have

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

However, since  $\langle a \rangle ff$  is the same as  $ff$  we get instead the sequence

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

which intuitively means that any processes belonging to approximation number  $k + 1$  satisfy  $\phi$  after a sequence of transitions of length at most  $k - 1$ .

## 4.2 Example

Recall the model for the readers and writers from Section 2.1. If we by “something good will eventually happens” mean that one of the writers (or equally some of the readers) eventually is allowed to write (read) we may choose to let

$$\langle w1 \rangle tt \vee \langle w2 \rangle tt$$

be the instance of  $\phi$  in (4). Checking the liveness property 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)
```

**Exercise** Would the liveness property still hold if we require that eventually one of the writers in particular should be writing, say  $\phi = \langle w1 \rangle tt$  is replacing  $\phi$  in (4)? What about the readers, i.e. would instantiating  $\phi$  by  $\langle r1 \rangle tt$  make the liveness property (4) hold?

## 5 Readers-Writers Program

The implementation of the CCS model of the readers and writers in Section 2.1 may be realized in Java as presented in this section.

### 5.1 The main program

First we present the overall program creating the database and starting threads with readers and writers.

```
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);}
    }
}
```

## 5.2 The readers and writers

Next, we present the implementations of the readers and writers. A reader is implemented by

```
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){}
    }
}
```

and a writer is symmetrically implemented by

```
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){}
    }
}
```

### 5.3 The database

Finally, the database is implemented as a monitor by:

```
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;
    }

    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();
    }

    synchronized public void acqW(int name) throws InterruptedException {
        while (readLocks > 0 || writeLocked) wait();
        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();
    }
}
```

Notice how the notification principles of the database are implemented.

**Exercise** Explain the notification principles of the database.

**Exercise** Try to run the program `ReadersWriters.java`.

## 6 Strong Liveness

As it probably became clear from running the program `ReadersWriters.java` it may happen that certain interleavings of the program prohibit the writers from ever having the opportunity to write to the database.

Take for instance the following initial part of the (non-deterministic) output from a run

```
java ReadersWriters
Reader 0 got the lock
Reader 1 got the lock
Reader 0 reads 0
Reader 0 releases the lock
Reader 0 got the lock
Reader 1 reads 0
Reader 1 releases the lock
Reader 1 got the lock
Reader 0 reads 0
Reader 0 releases the lock
Reader 0 got the lock
Reader 1 reads 0
Reader 1 releases the lock
Reader 1 got the lock
Reader 0 reads 0
Reader 0 releases the lock
Reader 0 got the lock
Reader 1 reads 0
Reader 1 releases the lock
Reader 1 got the lock
Reader 1 reads 0
Reader 1 releases the lock
Reader 1 got the lock
Reader 0 reads 0
Reader 0 releases the lock
Reader 0 got the lock
```

From the program output it seems that the writers are starved and not ever given a fair chance to write to the database.

Actually we don't know for sure from running (and hence testing) the program whether the writers ever get a chance to write to the database or not, but the trace indicates that it indeed is a possibility, and inspection of the implementation of the monitor that implements the access to the database clearly reveals that there is such a danger.

**Exercise** Explain why the writers can be starved.

### 6.1 Strengthening the Liveness Property

The starvation of the writers is however not in conflict with the liveness property (4) because it only states that there is a reachable state (along some path) somewhere in the execution tree satisfying  $\phi$ .

In order to guarantee fairness in the sense that in any run of our program the writers will eventually be writing to the database we must specify a stronger progress property. In terms of the notion of a execution tree we must specify that any path in the execution tree satisfy  $\phi$ . Such a property can be specified as a minimal fixed point by

$$\mu X.(\phi \vee (\bigvee_{a \in \mathcal{A}} \langle a \rangle tt \wedge \bigwedge_{a \in \mathcal{A}} [a]X)) \quad \text{Strong Liveness} \quad (5)$$

Intuitively the meaning of the formula is that either  $\phi$  holds in the current state, or some action in  $\mathcal{A}$  guarantees progress ( $\bigvee_{a \in \mathcal{A}} \langle a \rangle tt$ ) such no matter how progress is ensured then the formula holds recursively ( $\bigwedge_{a \in \mathcal{A}} [a]X$ ).

The reason why (5) is defined as a minimal fixed point is similar to the reason why (4) is defined as a minimal fixed point. For instance, taking  $P = a.P$  and  $\phi$  to be  $\langle b \rangle tt$  would make  $P$  satisfy the maximal fixed point formula

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

although  $P$  cannot perform  $b$ .

The maximal solution is the limit to

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

whereas the minimal solution is the limit to

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

In contrast to the maximal solution the minimal requires  $\phi$  to be satisfied at some finite level on each path.

## 6.2 Example

In this example we want to check as to whether the model of the readers and writers specified in Section 2.1 satisfy the strong liveness property (5) in which  $\phi$  is substituted by  $\langle w1 \rangle tt \vee \langle w2 \rangle tt$ .

However, first we make a drawing of the labelled transition system for the model of the readers and writers system

**Exercise** Make a drawing of the labelled transition system for the model of the readers and writers defined in Section 2.1. Argue why it doesn't satisfy (5) if we instantiate  $\phi$  by  $\langle w1 \rangle tt \vee \langle w2 \rangle tt$ .

In the dual case, if we in MWB check whether the readers are ever ensured the opportunity to write, say  $\phi$  is instantiated by  $\langle r1 \rangle tt \vee \langle r2 \rangle tt$  we get

```

MWB>prove Sys<r1,r2,w1,w2> mu X.( <r1>TT | <r2>TT
      | ( ( <t>TT | <r1>TT | <r2>TT | <w1>TT | <w2>TT )
          & ( [t]X & [r1]X & [r2]X & [w1]X & [w2]X ) ) )
Model Prover says: NO.
(409 inferences)

```

Hence the model of the readers and writers does not guarantee strong fairness in that both readers and writers may be starved, i.e. there exists paths where either readers or writers are not allowed to access the database.

### 6.3 Disclaimer

Unfortunately the subtool handling call to `prove` in MWB contains an error<sup>1</sup> in that if we check the strong liveness property from the exercise above we get the wrong answer:

```

MWB>prove Sys<r1,r2,w1,w2> mu X.( <w1>TT | <w2>TT
      | ( ( <t>TT | <r1>TT | <r2>TT | <w1>TT | <w2>TT )
          & ( [t]X & [r1]X & [r2]X & [w1]X & [w2]X ) ) )
Model Prover says: YES!
(373 inferences)

```

Making use instead of the MWB `check` commando, although it requires a slightly different syntax, we get the right answer

```

check Sys<r1,r2,w1,w2>
  (mu X(w1,w2,r1,r2).( <x>(x=w1) | <x>(x=w2)
    | ( ( <t>TT | <x>(x=r1) | <x>(x=r2) | <x>(x=w1) | <x>(x=w2) )
        & ( [t]X(w1,w2,r1,r2) & [x](x#r1 | X(w1,w2,r1,r2))
            & [x](x#r2 | X(w1,w2,r1,r2)) & [x](x#w1 | X(w1,w2,r1,r2))
            & [x](x#w2 | X(w1,w2,r1,r2)) ) ) ) ) (w1,w2,r1,r2)
No. (87 inferences)

```

For an explanation of the syntax see [2].

## 7 Giving Priority to Writers

Before going into the discussion about fairness let us for a while assume that writers may be considered more important than readers. That is not always an unrealistic scenario since often readers may be interested in the most recent updates.

Of course an easy solution would be to simply give higher priority to writer threads than to the reader threads, however we shall not do so. Instead, we shall modify the database implementation provided in Section 5.3 to give priority to writers. The modification is an intermediary step toward the fair solution demonstrated in Section 8.

---

<sup>1</sup>Björn Victor, one of the developers of MWB, kindly made me aware of the fact that MWB contains known bugs and that a new version of MWB is going to be developed.

The idea used to make an implementation where priority is given to writers is to extend the lock protocol in the model of the readers and writers in Section 2.1 by letting the writers make a request before acquiring the lock and to prohibit readers to get the lock if writers are waiting, i.e. if writers have made a request.

## 7.1 Implementing writers priority

Next we go into the details with the implementation. We only show the code for the refined implementation of the database, since the other parts of the code is unchanged. The making of a request is implemented simply by incrementing a variable called `writeWaiting` when calling the method `acqW`.

```
public class UnfairDatabase {

    private int content = 0, readLocks = 0, writeWaiting = 0;
    private boolean writeLocked = false;

    synchronized public void write(int i) { content = i; }

    synchronized public int read() { return content; }

    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
    }

    synchronized public void acqW(int name) throws InterruptedException {
        writeWaiting++; // 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();
    }
}
```

**Exercise** Explain the modifications to the implementation of the database described in Section 5.3 as carried out in the code above. The modified lines are marked by A, B, C, and D.

## 7.2 A CCS model with prioritized writers

Recall the readers and writers model from Section 2.1. In this section we modify the model in order to make it reflect the program above. It turns out that it is sufficient to redo only the writer process and the lock.

The first change is that 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 changed in order to be able to handle also request from writers. In order to do so we associate the state of a lock with two indices. The first index indicates how many readers already have the lock and the second index tells how many writers are waiting for the lock. For instance, `Lock11 (reqW, acqW, relW, acqR, relR)` is the state where one reader have got the lock and where one writer is waiting for obtaining it.

Here is the full specification of the lock

```
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 complete system is defined to be

```
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> )
```

If we on this model try to validate the properties checked in Section 2.2 we get not surprisingly exactly the same result, i.e. no two writers must access the database simultaneously. and neither may a writer and a reader, whereas two readers are allowed to access the database concurrently.

On the other hand, using the model above we may now be fooled to expect that the strong liveness property is satisfied if we in (5) instantiate  $\phi$  by  $\langle w1 \rangle tt \vee \langle w2 \rangle tt$ .

However, that strong liveness property is not satisfied. The reason why is that there exists paths in the execution tree of the model where no writer ever issue a request for writing to the database, hence obviously there exists paths also where only readers repetitively acquire the lock all the time. Therefore there exists a path where no writers ever get a chance to write to the database.

In practice, for the implementation above, this phenomenon of being incapable of satisfying the strong fairness property would correspond to prohibit the writers ever entering the monitor implementing the database.

This writer monitor exclusion can be obtained by making the reader threads having priority over writer threads such that the scheduler always make a reader become the runnable thread, or by having sufficiently many readers always ensuring the monitor lock to be held by a reader.

If we disregard both of these two extreme situations, and hence assume fairness toward entering the monitor, then for all program runs the strong liveness property where eventually a writer is granted the access to the database can be considered to be satisfied.

## 8 Ensuring Fairness

If we continue under the assumption that all readers and writers are provided fair access to the database monitor we shall now pursue to establish a fair solution to the readers/writers problem, i.e. a solution where not only the writers but also the readers eventually are granted access to the database.

Based upon the solution in favour of the writers above we may define yet another solution where the only change is a boolean variable which value tells as to whether it is a readers or a writers turn to access the database.

Intuitively, the variable introduces asymmetry into the implementation such that readers cannot always be blocked because of waiting writers. In fact we should expect the implementation to satisfy both the liveness properties from Section 6.2.

We only show the refined implementation of the monitor.

```

public class FairDatabase {

    private int content = 0, readLocks = 0, writeWaiting = 0;
    private boolean writeLocked = false, readersTurn = true;

    synchronized public void write(int i) { content = i; }

    synchronized public int read() { return content; }

    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();
    }

    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();
    }
}

```

**Exercise** Try to run the program.

Below we give informal arguments as to why neither readers nor writers can be starved choosing this solution.

## 8.1 Why cannot readers and writers be starved?

Suppose a writer got the read/write lock and suppose several writers are waiting for the lock (i.e. they have incremented `writeWaiting`). Before the writer having obtained the lock releases it the turn is switched to belong to the readers before all waiting processes are notified. Hence no writer will be allowed to obtain the lock next and since `writeLocked` is false the condition for any of the readers to wait is not satisfied and therefore a reader is allowed to enter the monitor.

Suppose several readers have the read/write lock and that a writer is waiting for the lock, i.e. `writeWaiting` is greater than zero. When one of the readers releases the lock it sets the turn to belong to the writers. Hence no new readers will be allowed to acquire the lock because the condition `writeWaiting > 0 && !readersTurn` is satisfied. Eventually all readers have released the lock so `readLocks` becomes zero and a writer obtains the lock.

## 9 Individual Fairness

The implementations above ensures fairness in the sense from Section 6.2, that is some reader (writer) is eventually given the opportunity to read (write). However, one particular reader (writer) may never be given the chance, she may be *individually* starved.

In order to ensure individual fairness (still under the assumption of fair access to the monitor implementation of the database) we introduce a kind of FIFO ordering on the requests to the database. We don't literally introduce a FIFO queue but we do so indirectly by implementing a so called *ticketing principle*.

The idea is to introduce a sequence of tickets (the values of an integer variable) from which tickets consecutively are given to each reader and writer upon requesting the database based on a first come first serve principle. Then simply readers and writers are allowed to request the database in order of their ticket number.

The changes to the database is illustrated below where `ticket` holds the value of the next ticket to be issued whereas `next` holds the value of the ticket to be served next.

```

public class FifoDatabase {

    private int content = 0, readLocks = 0, ticket = 0, next = 0;

    synchronized public void write(int i) { content = i; }

    synchronized public int read() { return content; }

    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();
    }

    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();
    }
}

```

## 9.1 Why a first come first serve implementation?

Suppose a writer has got the write lock. Then no matter as to whether a reader or a writer has the next ticket it will only be served after `next` has been incremented in the writers call to `relW`.

Suppose a reader has got the read lock. Then if a reader has got the next ticket to be served then since `next` is incremented already in the `acqR` method the next reader may start reading immediately. If a writer has got the next ticket then she must wait writing until all readers have released their lock.

**Exercise** Try to run the program.

## 10 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  $\phi$  is  $\langle b \rangle tt$  although  $P$  can do no  $b$ .
4. Continue with the second mandatory assignment

## References

- [1] J.C. Godskesen. Deadlocks and safety properties (version 1.2), 2005. Teaching notes, IT University of Copenhagen.
- [2] B. Victor. The mobility workbench user's guide, polyadic version 3.122. Technical report, SICS, Stockholm, Sweden, 1995.