Safety and Liveness

Abhik Roychoudhury
CS 3211
National University of Singapore

From Kramer and Magee’s lecture notes.
Reading material: Chapter 7 of Textbook.

Concepts: properties: true for every possible execution
safety: nothing bad happens
liveness: something good eventually happens

Models:
safety: no reachable ERROR/STOP state
progress: an action is eventually executed
fair choice and action priority

Practice: threads and monitors

Aim: property satisfaction.

7.1 Safety

A safety property asserts that nothing bad happens.

- STOP or deadlocked state (no outgoing transitions)
- ERROR process (-1) to detect erroneous behavior

Safety properties

Property that it is polite to knock before entering a room.
Traces: knock→enter √ enter × knock→knock

property POLITE = (knock→enter→POLITE).

In all states, all the actions in the alphabet of a property are eligible choices.

Safety - property specification

- ERROR conditions state what is not required (cf. exceptions).
- in complex systems, it is usually better to specify safety properties by stating directly what is required.

property SAFE_ACTUATOR = (command → respond
→ SAFE_ACTUATOR).

Trace to ERROR:
command command

Safety properties

Safety property P defines a deterministic process that asserts that any trace including actions in the alphabet of P, is accepted by P.

Thus, if P is composed with S, then traces of actions in the alphabet of S ∩ alphabet of P must also be valid traces of P, otherwise ERROR is reachable.

Transparency of safety properties:
Since all actions in the alphabet of a property are eligible choices, composing a property with a set of processes does not affect their fairness behavior. However, if a behavior can occur which violates the safety property, then ERROR is reachable. Properties must be deterministic to be transparent.
Safety properties

How can we specify that some action, disaster, never occurs?

property CALM = STOP + \{disaster\}.

A safety property must be specified so as to include all the acceptable, valid behaviors in its alphabet.

Safety - mutual exclusion

\[ \text{LOOP} = (\text{mutex.down} \rightarrow \text{enter} \rightarrow \text{exit} \rightarrow \text{mutex.up} \rightarrow \text{LOOP}). \]
\[ \text{||SEMADEMO} = (p[1..3]:\text{LOOP} \rightarrow \text{mutex:SEMAPHORE}(1)). \]

How do we check that this does indeed ensure mutual exclusion in the critical section?

property MUTEX = (p[1..3].\text{enter} \rightarrow p[i].\text{exit} \rightarrow \text{MUTEX}).

||CHECK = (SEMADEMO || MUTEX).

Check safety using LTS.

What happens if semaphore is initialized to 2?

7.2 Single Lane Bridge problem

A bridge over a river is only wide enough to permit a single lane of traffic. Consequently, cars can only move concurrently if they are moving in the same direction. A safety violation occurs if two cars moving in different directions enter the bridge at the same time.

A bridge over a river is only wide enough to permit a single lane of traffic. Consequently, cars can only move concurrently if they are moving in the same direction. A safety violation occurs if two cars moving in different directions enter the bridge at the same time.

Single Lane Bridge - model

- Events or actions of interest?
  - enter and exit
- Identify processes.
  - cars and bridge
- Identify properties.
  - oneway
- Define each process and interactions (structure).

Single Lane Bridge - CONVOY model

\[ \text{NOPASS1} = C[1], \text{NOPASS2} = C[1]. \]
\[ \text{C[i]:ID} = ([i].\text{enter} \rightarrow C[1/2N+1]). \]
\[ \text{C[i]:ID} = ([i].\text{exit} \rightarrow C[1/2N+1]). \]
\[ \text{||CONVOY} = (||\text{CAR} \rightarrow \text{NOPASS1} \rightarrow \text{NOPASS2}). \]

Permits:
- 1.enter \rightarrow 2.enter \rightarrow 1.exit \rightarrow 2.exit
But not:
- 1.enter \rightarrow 2.enter \rightarrow 2.exit \rightarrow 1.exit
ie. no overtaking.

Single Lane Bridge - CARS model

\text{const N} = 3 \quad \text{// number of each type of car}
\text{range T = 0..N} \quad \text{// type of car count}
\text{range ID= 1..N} \quad \text{// car identities}

\text{CAR} = (\text{enter} \rightarrow \text{exit} \rightarrow \text{CAR}).

To model the fact that cars cannot pass each other on the bridge, we model a CONVOY of cars in the same direction. We will have a red and a blue convoy of up to N cars for each direction:

\[ \text{||CARS} = (\text{red:CONVOY} \rightarrow \text{blue:CONVOY}). \]
Single Lane Bridge - BRIDGE model
Cars can move concurrently on the bridge only if in the same direction. The bridge maintains counts of blue and red cars on the bridge. Red cars are only allowed to enter when the blue count is zero and vice-versa.

\[
\text{BRIDGE} = \text{BRIDGE}[0][0], \quad \text{BRIDGE}[nr:T][nb:T] =
\]
\[
\begin{align*}
\text{red[ID].enter} &\rightarrow \text{BRIDGE}[nr+1][nb] \quad \text{when}(nb==0) \\
\text{blue[ID].enter} &\rightarrow \text{BRIDGE}[nr][nb+1] \quad \text{when}(nr==0)
\end{align*}
\]

Even when O, exit actions permit the car counts to be decremented.

Single Lane Bridge - model analysis
| | | SingleLaneBridge = (CARS | BRIDGE | ONEWAY). |
|---| | Is the safety property ONEWAY violated? |
| | | No deadlocks/errors |
| | | Without the BRIDGE constraints, is the safety property ONEWAY violated? |
| Trace to property violation in ONEWAY: |
| | | red.1.enter blue.1.enter |

Single Lane Bridge - BridgeCanvas
An instance of BridgeCanvas class is created by SingleLaneBridge applet - ref is passed to each newly created RedCar and BlueCar object.

```java
class BridgeCanvas extends Canvas {
    public void init(int ncars) {…} //set number of cars
    public boolean moveRed(int i) throws InterruptedException {…} //move red car with the identity i a step
    public boolean moveBlue(int i) throws InterruptedException {…} //move blue car with the identity i a step
    public synchronized void freeze(){…} // freeze display
    public synchronized void thaw(){…} //unfreeze display
}
```

Single Lane Bridge - RedCar
```java
class RedCar implements Runnable {
    BridgeCanvas display; Bridge control; int id;
    RedCar(Bridge b, BridgeCanvas d, int id) {
        display = d; this.id = id; control = b;
    }
    public void run() {
        try{
            while(true) {
                while (!display.moveRed(id)); // not on bridge
                control.redEnter(); // request access to bridge
                while (display.moveRed(id)); // move over bridge
                control.redExit(); // release access to bridge
            }
            catch (InterruptedException e) {} }
    }
}
```

Similarly for the BlueCar.
Single Lane Bridge - class Bridge

```java
class Bridge {
    synchronized void redEnter() throws InterruptedException {}
    synchronized void redExit() {}
    synchronized void blueEnter() throws InterruptedException {}
    synchronized void blueExit() {}
}
```

Class Bridge provides a null implementation of the access methods i.e. no constraints on the access to the bridge.

Result........... ?

Single Lane Bridge - SafeBridge

```java
class SafeBridge extends Bridge {
    private int nred = 0; //number of red cars on bridge
    private int nblue = 0; //number of blue cars on bridge
    // Monitor Invariant: nred≥0 and nblue≥0 and
    // not (nred>0 and nblue>0)
    synchronized void redEnter() throws InterruptedException {
        while (nblue>0) wait();
        ++nred;
    }
    synchronized void redExit() {
        --nred;
        if (nred==0) notifyAll();
    }
    synchronized void blueEnter() throws InterruptedException {
        while (nred>0) wait();
        ++nblue;
    }
    synchronized void blueExit() {
        --nblue;
        if (nblue==0) notifyAll();
    }
}
```

This is a direct translation from the BRIDGE model.

To avoid unnecessary thread switches, we use conditional notification to wake up waiting threads only when the number of cars on the bridge is zero i.e. when the last car leaves the bridge.

But does every car eventually get an opportunity to cross the bridge? This is a liveness property.

7.3 Liveness

A safety property asserts that nothing bad happens.
A liveness property asserts that something good eventually happens.

Single Lane Bridge: Does every car eventually get an opportunity to cross the bridge?

ie. make PROGRESS?

A progress property asserts that it is always the case that an action is eventually executed. Progress is the opposite of starvation, the name given to a concurrent programming situation in which an action is never executed.

Progress properties - fair choice

Fair Choice: If a choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often.

If a coin were tossed an infinite number of times, we would expect that heads would be chosen infinitely often and that tails would be chosen infinitely often.

This requires Fair Choice!
Progress properties

\[ \text{progress } P = \{a_1, a_2, \ldots, a_n\} \text{ defines a progress property } P \text{ which asserts that in an infinite execution of a target system, at least one of the actions } a_1, a_2, \ldots, a_n \text{ will be executed infinitely often.} \]

\[ \text{COIN system: } \begin{align*}
\text{progress HEADS} &= \{\text{heads}\} \\
\text{progress TAILS} &= \{\text{tails}\}
\end{align*} \]

\[ \text{LTSA check progress: } \text{No progress violations detected.} \]

Suppose that there were two possible coins that could be picked up:

- A trick coin
- A regular coin

\( \text{TWO} = (\text{pick} \rightarrow \text{COIN} \mid \text{pick} \rightarrow \text{TRICK}), \)
\( \text{TRICK} = (\text{toss} \rightarrow \text{heads} \rightarrow \text{TRICK}), \)
\( \text{COIN} = (\text{toss} \rightarrow \text{heads} \rightarrow \text{COIN} \mid \text{toss} \rightarrow \text{tails} \rightarrow \text{COIN}). \)

\( \text{TWOCOIN: } \begin{align*}
\text{progress HEADS} &= \{\text{heads}\} \\
\text{progress TAILS} &= \{\text{tails}\}
\end{align*} \)

Progress analysis

A progress property is violated if analysis finds a terminal set of states in which none of the progress set actions appear.

\[ \text{progress TAILS} = \{\text{tails}\} \text{ in } \{1, 2\} \]

Default analysis for \( \text{TWO} \): separate progress property for every action.

\[ \text{Default for } \text{TWO} \text{COIN: } \begin{align*}
\text{progress } P &= \{\text{pick}\} \\
\text{progress } P &= \{\text{toss, heads, tails}\}
\end{align*} \]

Progress violation for actions:
- \{pick\}
- \{toss, heads, tails\}

If the default holds, then every other progress property holds i.e. every action is executed infinitely often and system consists of a single terminal set of states.
The Single Lane Bridge implementation can permit progress violations. However, if default progress analysis is applied to the model then no violations are detected.

**Why not?**

For choice means that eventually every possible execution occurs, including those in which cars do not starve. To detect progress problems we must check under adverse conditions. We superimpose some scheduling policy for actions, which models the situation in which the bridge is congested.

Progress - action priority

Action priority expressions describe scheduling properties:

**High Priorit y (<<)**
\[ |C = (P|Q) << \{a_1, \ldots, a_n\} \]

specifies a composition in which the actions \(a_1, \ldots, a_n\) have higher priority than any other action in the alphabet of \(P|Q\) including the silent action \(\tau\). In any choice in this system which has one or more of the actions \(a_1, \ldots, a_n\) labeling a transition, the transitions labeled with lower priority actions are discarded.

**Low Priorit y (>>)**
\[ |C = (P|Q) >> \{a_1, \ldots, a_n\} \]

specifies a composition in which the actions \(a_1, \ldots, a_n\) have lower priority than any other action in the alphabet of \(P|Q\) including the silent action \(\tau\). In any choice in this system which has one or more transitions not labeled by \(a_1, \ldots, a_n\), the transitions labeled by \(a_1, \ldots, a_n\) are discarded.

7.4 Congested single lane bridge

progress BLUECROSS = \{blue[ID].enter\}
progress REDCROSS = \{red[ID].enter\}

BLUECROSS - eventually one of the blue cars will be able to enter
REDCROSS - eventually one of the red cars will be able to enter

Congestion using action priority?

Could give red cars priority over blue (or vice versa)? Instead we merely encourage congestion by lowering the priority of the exit actions of both cars from the bridge.

Progress Analysis? LTS?

congested single lane bridge model

This corresponds with the observation that, with more than one car, it is possible that whichever color car enters the bridge first will continuously occupy the bridge preventing the other color from ever crossing.
The bridge needs to know whether or not cars are waiting to cross.

Modify CAR:

\[\text{CAR} = (\text{request}\rightarrow\text{enter}\rightarrow\text{exit}\rightarrow\text{CAR}).\]

Modify BRIDGE:

Red cars are only allowed to enter the bridge if there are no blue cars on the bridge and there are no blue cars waiting to enter the bridge.

Blue cars are only allowed to enter the bridge if there are no red cars on the bridge and there are no red cars waiting to enter the bridge.

/* nr – number of red cars on the bridge
   wb – number of blue cars waiting to enter
   wr – number of red cars waiting to enter
   nb – number of blue cars on the bridge
*/

\[
\text{BRIDGE} = \text{BRIDGE}[0][0][0][0],
\]

\[
\text{BRIDGE}[nr:T][nb:T][wr:T][wb:T] =
\]

\[
\begin{align*}
&\text{red[ID].request }\rightarrow \text{BRIDGE}[nr][nb][wr+1][wb] \\
&\text{when (nb==0 && wr==0)}
\end{align*}
\]

\[
\text{d[ID]}\quad \text{BRIDGE}[1][b][1][b]
\]

\[
\begin{align*}
&\text{red[ID].enter }\rightarrow \text{BRIDGE}[nr+1][nb][wr-1][wb] \\
&\text{red[ID].exit }\rightarrow \text{BRIDGE}[nr-1][nb][wr][wb][True] \\
&\text{blue[ID].request }\rightarrow \text{BRIDGE}[nr][nb][wr][wb+1] \\
&\text{when (nr==0 && wr==0)}
\end{align*}
\]

\[
\begin{align*}
&\text{blue[ID].enter }\rightarrow \text{BRIDGE}[nr][nb+1][wr][wb-1] \\
&\text{blue[ID].exit }\rightarrow \text{BRIDGE}[nr][nb-1][wr][wb][False] \\
&\text{blue[ID].exit }\rightarrow \text{BRIDGE}[nr][nb-1][wr][wb][False]
\end{align*}
\]

OK now?

This is a direct translation from the model.
A shared database is accessed by two kinds of processes. Readers execute transactions that examine the database while Writers both examine and update the database. A Writer must have exclusive access to the database; any number of Readers may concurrently access it.

**readers/writers model**

- **Events or actions of interest?**
  acquireRead, releaseRead, acquireWrite, releaseWrite
- **Identify processes.**
  Readers, Writers & the RW_Lock
- **Identify properties.**
  RW_Safe
  RW_Progress
- **Define each process and interactions (structure).**

**readers/writers model - READER & WRITER**

- **set Actions =**
  (acquireRead, releaseRead, acquireWrite, releaseWrite)
- **READER =**
  (acquireRead -> examine -> releaseRead -> READER) + (actions)
- **WRITER =**
  (acquireWrite -> modify -> releaseWrite -> WRITER) + (actions)

**readers/writers model - RW_Lock**

```
const False = 0  const True  = 1
range Bool  = False..True
const Nread = 2   // Maximum readers
const Nwrite = 2  // Maximum writers
RW_LOCK = RW[0][False],
          RW[readers][writing]:Bool
acquireRead  -> RW[readers+1][writing]
|releaseRead      -> RW[readers-1][writing]
|when (readers==0 && !writing) acquireWrite -> RW[readers][True]
|releaseWrite     -> RW[readers][False]
```

The lock maintains a count of the number of readers, and a Boolean for the writers.

**readers/writers model - safety**

```
property SAFE_RW
   = (acquireRead -> READING[1]
      | acquireWrite -> WRITING
   )
   READING[i:1..Nread]
   = (acquireRead -> READING[i+1]
      | when(i>1) releaseRead -> READING[i-1]
      | when(i=1) releaseRead -> SAFE_RW
   )
   WRITING = (releaseWrite -> SAFE_RW).
```

We can check that RW_LOCK satisfies the safety property.

```
||READWRITELOCK = (RW_LOCK || SAFE_RW).
```

**readers/writers model**

An ERROR occurs if a reader or writer is badly behaved (release before acquire or more than two readers).

We can now compose the READWRITELOCK with READER and WRITER processes according to our structure...
readers/writers - progress

progress WRITE = {writer[1..Nwrite].acquireWrite}
progress READ = {reader[1..Nread].acquireRead}

WRITE - eventually one of the writers will acquireWrite
READ - eventually one of the readers will acquireRead

Adverse conditions using action priority?
we lower the priority of the release actions for both readers and writers.

||RW_PROGRE = READERS_WRITERS
>>{reader[1..Nread].releaseRead,
writer[1..Nwrite].releaseWrite}.

Progress Analysis ? LTS?

readers/writers implementation - monitor interface

We concentrate on the monitor implementation:

```
interface ReadWrite {
    public void acquireRead()
        throws InterruptedException;
    public void releaseRead();
    public void acquireWrite()
        throws InterruptedException;
    public void releaseWrite();
}
```

We define an interface that identifies the monitor methods that must be implemented, and develop a number of alternative implementations of this interface.

Firstly, the safe READWRITELOCK.

readers/writers implementation - ReadWriteSafe

```
public synchronized void acquireWrite()
    throws InterruptedException {
    while (writing)
        wait();
    writing = true;
}
```

```
public synchronized void releaseWrite() {
    writing = false;
    notifyAll();
}
```

```
Unblock all readers
```

However, this monitor implementation suffers from the WRITE progress problem: possible writer starvation if the number of readers never drops to zero.

Solution?

readers/writers - writer priority

```
set Actions = {acquireRead, releaseRead, acquireWrite, releaseWrite, requestWrite}
```

```
WRITER = {requestWrite->acquireWrite->modify
    ->acquireWrite->WRITER
    }+Actions\{modify\}.
```

```
Strategy: Block readers if there is a writer waiting.
```
readers/writers model - writer priority

{\textbf{RW\_LOCK} = RW[0][False][0],
RW[readers:0..Nread][writing:Bool][waitingW:0..Nwrite] =
{\begin{align*}
\text{when} \ (\text{writing} \ &\& \ \text{waitingW}=0) \\
\quad \text{acquireRead} &\rightarrow RW[\text{readers+1}][\text{writing}][\text{waitingW}] \\
\quad \text{releaseRead} &\rightarrow RW[\text{readers-1}][\text{writing}][\text{waitingW}] \\
\text{when} \ (\text{readers}=0 \ &\& \ \text{writing}) \\
\quad \text{acquireWrite} &\rightarrow RW[\text{readers}][\text{True}][\text{waitingW}-1] \\
\quad \text{releaseWrite} &\rightarrow RW[\text{readers}][\text{False}][\text{waitingW}] \\
\quad \text{requestWrite} &\rightarrow RW[\text{readers}][\text{writing}][\text{waitingW}+1] \\
{\end{align*}}
\}

\textbf{Safety and Progress Analysis ?}

\begin{itemize}
\item \textbf{Safety}:
\begin{itemize}
\item no deadlocks/errors
\end{itemize}
\item \textbf{Progress}: READ and WRITE:
\begin{itemize}
\item Progress violation: READ
\begin{itemize}
\item \text{Path to terminal set of states:}
\item \text{writer.1.requestWrite, writer.2.requestWrite}
\item \text{Actions in terminal set:}
\item \text{writer.1.requestWrite, writer.1.acquireWrite, writer.2.requestWrite, writer.2.acquireWrite, writer.2.releaseWrite}
\end{itemize}
\end{itemize}
\end{itemize}

In practice, this may be satisfactory as there is usually more read access than write, and readers generally want the most up to date information.

readers/writers model - writer priority

\begin{itemize}
\item \textbf{Safety}:
\begin{itemize}
\item no deadlocks/errors
\end{itemize}
\item \textbf{Progress}: READ and WRITE:
\begin{itemize}
\item Progress violation: READ
\begin{itemize}
\item \text{Path to terminal set of states:}
\item \text{writer.1.requestWrite, writer.2.requestWrite}
\item \text{Actions in terminal set:}
\item \text{writer.1.requestWrite, writer.1.acquireWrite, writer.2.requestWrite, writer.2.acquireWrite, writer.2.releaseWrite}
\end{itemize}
\end{itemize}
\end{itemize}

Readers starvation: it is always a writer waiting.

\begin{itemize}
\item \text{May also be readers waiting}
\end{itemize}

readers/writers implementation - ReadWritePriority

class ReadWritePriority implements ReadWrite{
private int readers =0;
private boolean writing = false;
private int waitingW = 0;
// no of waiting Writers.

public synchronized void acquireRead() throws InterruptedException {
while (writing || waitingW>0) wait();
++readers;
}

public synchronized void releaseRead() {
--readers;
if(readers==0) notifyAll();
}

readers/writers implementation - ReadWritePriority

synchronized public void acquireWrite() throws InterruptedException {
++waitingW;
while (readers>0 || writing) wait();
--waitingW;
writing = true;
}
synchronized public void releaseWrite() {
writing = false;
notifyAll();
}

Both READ and WRITE progress properties can be satisfied by introducing a turn variable as in the Single Lane Bridge.

Summary

\begin{itemize}
\item \textbf{Concepts}:
\begin{itemize}
\item properties: true for every possible execution
\item safety: nothing bad happens
\item liveness: something good eventually happens
\end{itemize}
\item \textbf{Models}:
\begin{itemize}
\item safety: no reachable ERROR/STOP state
\item progress: compose safety properties at appropriate stages
\item on an action is eventually executed
\item action is eventually executed
\item fair choice and action priority
\item progress check on the final system model
\end{itemize}
\item \textbf{Practice}:
\begin{itemize}
\item threads and monitors
\end{itemize}
\end{itemize}