safety & liveness properties

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

Models:  safety: no reachable \texttt{ERROR/STOP} state
           progress: an action is \textit{eventually} executed
           fair choice and action priority

Practice: threads and monitors

\textbf{Aim}: property satisfaction.
10.1 Safety

*A safety property asserts that nothing bad happens.*

♦ **STOP** or deadlocked state (no outgoing transitions)

♦ **ERROR** process (-1) to detect erroneous behaviour

ACTUATOR

\[
\begin{align*}
\text{ACTUATOR} & = (\text{command} \rightarrow \text{ACTION}), \\
\text{ACTION} & = (\text{respond} \rightarrow \text{ACTUATOR} | \text{command} \rightarrow \text{ERROR}).
\end{align*}
\]

♦ analysis using LTSA: (shortest trace)

**Trace to ERROR:**

\[
\begin{align*}
\text{command} & \\
\text{command}
\end{align*}
\]
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.

```plaintext
property SAFE_ACTUATOR = (command -> respond -> SAFE_ACTUATOR).
```

- Analysis using **LTSA** as before.
Safe Actuator

\[
\text{ACTUATOR} = (\text{command} \rightarrow \text{ACTION}), \\
\text{ACTION} = (\text{respond} \rightarrow \text{ACTUATOR} \mid \text{command} \rightarrow \text{ACTUATOR}).
\]

property SAFE_ACTUATOR
\[
= (\text{command} \rightarrow \text{respond} \rightarrow \text{SAFE_ACTUATOR}).
\]

||CHECK = (\text{ACTUATOR} || \text{SAFE_ACTUATOR})

property SAFE_ACTUATOR violation
Trace to property violation in SAFE_ACTUATOR
command
command
Safety properties

Property that it is polite to knock before entering a room.

Traces: \( \text{knock} \rightarrow \text{enter} \quad \checkmark \quad \text{enter} \)

\( \text{knock} \rightarrow \text{knock} \)

\textbf{property} \ \text{POLITE} \quad = (\text{knock} \rightarrow \text{enter} \rightarrow \text{POLITE}) .

In all states, all the actions in the alphabet of a property are eligible choices.
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 $S \cap$ 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 correct 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?

\[
\text{property } \text{CALM} = \text{STOP} + \{\text{disaster}\}.
\]

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

A safety property asserts that nothing bad happens.

A liveness property asserts that something good eventually happens.

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**!

\[
COIN = (\text{toss} \rightarrow \text{heads} \rightarrow COIN \mid \text{toss} \rightarrow \text{tails} \rightarrow COIN).
\]
Progress properties

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

**COIN system:**
progress HEADS = \{heads\}  
progress TAILS = \{tails\}

**LTSA check progress:**
No progress violations detected.
Progress properties

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

a trick coin and a regular coin......

\[
\text{TWO}COIN = (\text{pick} \rightarrow \text{COIN}|\text{pick} \rightarrow \text{TRICK}),
\]

\[
\text{TRICK} = (\text{toss} \rightarrow \text{heads} \rightarrow \text{TRICK}),
\]

\[
\text{COIN} = (\text{toss} \rightarrow \text{heads} \rightarrow \text{COIN}|\text{toss} \rightarrow \text{tails} \rightarrow \text{COIN}).
\]

\[
\text{TWO}COIN: \quad \text{progress HEADS} = \{\text{heads}\} \quad \checkmark
\]

\[
\text{progress TAILS} = \{\text{tails}\} \quad \times
\]

CSC321 §10 Safety & Liveness Properties
Progress properties

\[
\text{progress HEADS} = \{\text{heads}\}
\]

\[
\text{progress TAILS} = \{\text{tails}\}
\]

**LTSA** check progress

**Progress violation: TAILS**

Path to terminal set of states:
- pick

Actions in terminal set:
- \{toss, heads\} 

\[
\text{progress HEADSorTails} = \{\text{heads,tails}\}
\]
Progress analysis

A terminal set of states is one in which every state is reachable from every other state in the set via one or more transitions, and there is no transition from within the set to any state outside the set.

Terminal sets for \textsc{TwoCoin}:

\{1,2\} and \{3,4,5\}

Given \textit{fair choice}, each terminal set represents an execution in which each action used in a transition in the set is executed infinitely often.

Since there is no transition out of a terminal set, any action that is \textit{not} used in the set cannot occur infinitely often in all executions of the system - and hence represents a \textit{potential progress violation}!
**Progress analysis**

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

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

**Default**: given fair choice, for *every* action in the alphabet of the target system, that action will be executed infinitely often. This is equivalent to specifying a separate progress property for every action.

Default analysis for **TWOCOIN**?
Progress analysis

Default analysis for TWOCOIN: separate progress property for every action.

Progress violation for actions: \{pick\}
Path to terminal set of states: pick
Actions in terminal set: \{toss, heads, tails\}

Progress violation for actions: \{pick, tails\}
Path to terminal set of states: pick
Actions in terminal set: \{toss, heads\}

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.
Progress - action priority

Action priority expressions describe scheduling properties:

High Priority ("<<")

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

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 Priority (">>")

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

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.
Progress - action priority

\[
\text{NORMAL} = (\text{work} \rightarrow \text{play} \rightarrow \text{NORMAL} \mid \text{sleep} \rightarrow \text{play} \rightarrow \text{NORMAL}).
\]

Action priority simplifies the resulting LTS by discarding lower priority actions from choices.

\[
\mid | \text{HIGH} = (\text{NORMAL}) \ll \{\text{work}\}.
\]

\[
\mid | \text{LOW} = (\text{NORMAL}) \gg \{\text{work}\}.
\]
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
   \ {examine}.

WRITER = (acquireWrite→modify→releaseWrite→WRITER)
   + Actions
   \ {modify}.

**Alphabet extension** is used to ensure that the other access actions cannot occur freely for any prefixed instance of the process (as before).

**Action hiding** is used as actions `examine` and `modify` are not relevant for access synchronisation.
The readers/writers model - **RW_LOCK**

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

```plaintext
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:0..Nread][writing:Bool] =
  (when (!writing)
   acquireRead  -> RW[readers+1][writing]
   |releaseRead  -> RW[readers-1][writing]
   |when (readers==0 && !writing)
            acquireWrite  -> RW[readers][True]
   |releaseWrite -> RW[readers][False]
  ).
```
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).

Safety Analysis? LTS?
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... …

\[
\text{READERS_WRITERS} = (\text{reader}[1..Nread] : \text{READER} \\
\quad \quad || \quad \text{writer}[1..Nwrite] : \text{WRITER} \\
\quad \quad || \quad \{\text{reader}[1..Nread], \\
\quad \quad \quad \text{writer}[1..Nwrite]\} :: \text{READWRITELOCK}).
\]
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_PROGRESS = READERS_WRITERS
               >>\{reader[1..Nread].releaseRead,
                   writer[1..Nread].releaseWrite\}.

Progress Analysis ? LTS?
Progress violation: WRITE
Path to terminal set of states:
reader.1.acquireRead
Actions in terminal set:
{reader.1.acquireRead, reader.1.releaseRead, reader.2.acquireRead, reader.2.releaseRead}

Writer starvation:
The number of readers never drops to zero.

Try the Applet!
readers/writers implementation - monitor interface

We concentrate on the monitor implementation:

```java
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**

```java
class ReadWriteSafe implements ReadWrite {
    private int readers = 0;
    private boolean writing = false;

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

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

*Unblock a single writer when no more readers.*
readers/writers implementation - **ReadWriteSafe**

```java
public synchronized void acquireWrite() throws InterruptedException {
    while (readers > 0 || writing) wait();
    writing = true;
}

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

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**

**Strategy:**
Block readers if there is a writer waiting.

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

WRITER = (requestWrite\rightarrow acquireWrite\rightarrow modify
\rightarrow releaseWrite\rightarrow WRITER
)+Actions\{modify\}.
```

CSC321 §10 Safety & Liveness Properties
readers/writers model - writer priority

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

property RW_SAFE:

No deadlocks/errors

progress READ and WRITE:

Progress violation: READ
Path to terminal set of states:
  writer.1.requestWrite
  writer.2.requestWrite
Actions in terminal set:
  {writer.1.requestWrite, writer.1.acquireWrite, writer.1.releaseWrite, writer.2.requestWrite, writer.2.acquireWrite, writer.2.releaseWrite}

In practice, this may be satisfactory as is usually more read access than write, and readers generally want the most up to date information.
readers/writers implementation - **ReadWritePriority**

```java
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) notify();
    }
}
```
Both *READ* and *WRITE* progress properties can be satisfied by introducing a *turn* variable.

```java
synchronized public void acquireWrite() {
    ++waitingW;
    while (readers>0 || writing) try{ wait();} catch(InterruptedException e){}
    --waitingW;
    writing = true;
}

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

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

◆ Models
  ● safety: no reachable ERROR/STOP state
    compose safety properties at appropriate stages
  ● progress: an action is eventually executed
    fair choice and action priority
    apply progress check on the final target system model

◆ Practice
  ● threads and monitors
  
  **Aim:** property satisfaction