3. Safety and Synchronization

Prof. O. Nierstrasz

Selected material © Magee and Kramer
Roadmap

> Modelling interaction in FSP
> Safety — synchronizing critical sections
> Locking for atomicity
> The busy-wait mutual exclusion protocol
> Conditional synchronization
> Checking Safety properties
Roadmap

> Modelling interaction in FSP
> Safety — synchronizing critical sections
> Locking for atomicity
> The busy-wait mutual exclusion protocol
> Conditional synchronization
> Checking Safety properties
Modelling interaction — shared actions

Actions that are common between two processes are shared and can be used to model process interaction:

> Unshared actions may be arbitrarily interleaved
> Shared actions occur simultaneously for all participants

MAKER = ( make -> ready -> MAKER ).
USER = ( ready -> use -> USER ).

| | MAKER_USER = ( MAKER | | USER ).

What are the states of the LTS?
The traces?
Modelling interaction — handshake

A **handshake** is an action that signals acknowledgement

\[
\begin{align*}
\text{MAKER}_v2 &= ( \text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKER}_v2 ) . \\
\text{USER}_v2 &= ( \text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USER}_v2 ) . \\
\text{MAKER}_v2 \text{ USER}_v2 &= ( \text{MAKER}_v2 \text{ USER}_v2 ) .
\end{align*}
\]

*What are the states and traces of the LTS?*
Roadmap

- Modelling interaction in FSP
- **Safety — synchronizing critical sections**
- Locking for atomicity
- The busy-wait mutual exclusion protocol
- Conditional synchronization
- Checking Safety properties
Safety problems

Objects must only be accessed when they are in a **consistent state**, formalized by a **class invariant**.

Each method **assumes** the class invariant holds when it starts, and it **re-establishes** it when done.

*If methods interleave arbitrarily, an inconsistent state may be accessed, and the object may be left in a “dirty” state.*

Where shared resources are updated may be a **critical section**.
A race condition exists if safety may be violated by bad timing

```java
public class AccountBAD extends Account {
    // unsynchronized!
    public void withdraw(int amount) {
        while (amount > assets) {
            Thread.yield(); // busy wait
        }
        Thread.yield(); // race condition!
        assets -= amount;
        checkInvariant(); // might fail!
    }
}
```

if (! (assets >= 0)) { errors++; }
Interference

Consider the two processes:

\[
\begin{align*}
\{ x = 0 \} \\
\text{AInc:} & \quad x := x+1 \\
\text{BInc:} & \quad x := x+1 \\
\{ x = ? \}
\end{align*}
\]

How can these processes interfere?
Atomic actions

Individual reads and writes may be atomic actions:

\[
\begin{align*}
\text{const } N & = 3 \\
\text{range } T & = 0..N \\
\text{Var} & = \text{Var}[0], \\
\text{Var}[u:T] & = ( \text{read}[u] \rightarrow \text{Var}[u] \\
& \quad \mid \text{write}[v:T] \rightarrow \text{Var}[v] ). \\
\text{set } \text{VarAlpha} & = \{ \text{read}[T], \text{write}[T] \} \\
\text{Inc} & = ( \text{read}[v:0..N-1] \\
& \quad \rightarrow \text{write}[v+1] \\
& \quad \rightarrow \text{STOP} ) \quad +\text{VarAlpha}.
\end{align*}
\]
A single sequential thread requires no synchronization:

\[ \text{SeqInc} = (\text{Var} | | \text{Inc}). \]
Modelling interaction in FSP
Safety — synchronizing critical sections
**Locking for atomicity**
The busy-wait mutual exclusion protocol
Conditional synchronization
Checking Safety properties
Concurrent behaviour

*Without synchronization, concurrent threads may interfere:*

\[ \text{ParInc} = (\{a,b\}::\text{Var} | | a::\text{Inc} | | b::\text{Inc}). \]
Locking

Locks are used to make a critical section atomic:

\[
\begin{align*}
\text{LOCK} &= ( \text{acquire} \rightarrow \text{release} \rightarrow \text{LOCK} ). \\
\text{INC} &= ( \text{acquire} \\
&\quad \rightarrow \text{read}[v:0..N-1] \\
&\quad \rightarrow \text{write}[v+1] \\
&\quad \rightarrow \text{release} \\
&\quad \rightarrow \text{STOP} ) +\text{VarAlpha}.
\end{align*}
\]
Processes can synchronize critical sections by sharing a lock:

\[
ParInc2 = (\{a,b\}::VAR \mid \{a,b\}::LOCK \mid a:INC \mid b:INC).
\]
Synchronization in Java

Java Threads also synchronize using locks:

```java
synchronized T m() {
    // method body
}
```

is just convenient syntax for:

```java
T m() {
    synchronized (this) {
        // method body
    }
}
```

*Every object has a lock*, and Threads may use them to synchronize with each other.
Safety and Synchronization

Roadmap

- Modelling interaction in FSP
- Safety — synchronizing critical sections
- Locking for atomicity
- The busy-wait mutual exclusion protocol
- Conditional synchronization
- Checking Safety properties
Busy-Wait Mutual Exclusion Protocol

P1 sets enter\(_1\) := true when it wants to enter its CS, but sets turn := “P2” to yield priority to P2.

```
process P1
    loop
        enter1 := true
        turn := “P2”
        while enter2 and turn = “P2”
            do skip
        Critical Section
            enter1 := false
        Non-critical Section
    end
end

process P2
    loop
        enter2 := true
        turn := “P1”
        while enter1 and turn = “P1”
            do skip
        Critical Section
            enter2 := false
        Non-critical Section
    end
end
```

Is this protocol correct? Is it fair? Deadlock-free?
Atomic read and write

We can model integer and boolean variables as processes with atomic read and write actions:

```
range T = 1..2

Var = Var[1],
Var[u:T] =
    ( read[u] -> Var[u]
    | write[v:T] -> Var[v])

set Bool = {true,false}

BOOL(Init='false) = BOOL[Init],
BOOL[b:Bool] =
    ( is[b] -> BOOL[b]
    | setTo[x:Bool] -> BOOL[x])
```
Modelling the busy-wait protocol

Each process performs two actions in its CS:

P1 = ( enter1.setTo['true']
   -> turn.write[2]
   -> Gd1),
Gd1 =
   ( enter2.is['false'] -> CS1
    | enter2.is['true'] ->
      ( turn.read[1] -> CS1
       | turn.read[2] -> Gd1)),
CS1 = ( a -> b
   -> enter1.setTo['false']
   -> P1).

P2 = ( enter2.setTo['true']
   -> turn.write[1]
   -> Gd2),
Gd2 =
   ( enter1.is['false'] -> CS2
    | enter1.is['true'] ->
      ( turn.read[2] -> CS2
       | turn.read[1] -> Gd2)),
CS2 = ( c -> d
   -> enter2.setTo['false']
   -> P2).

||BusyWait = (enter1:BOOL|enter2:BOOL|turn:Var|P1|P2)@{a,b,c,d}.
Busy-wait composition

Very pretty, but how do we know there are no errors?!
Checking for errors

We can check for errors by composing our system with an agent that moves to the ERROR state if atomicity is violated:

\[
\text{Ok} = (a \rightarrow (c \rightarrow \text{ERROR} \mid b \rightarrow \text{Ok}) \\
\mid c \rightarrow (a \rightarrow \text{ERROR} \mid d \rightarrow \text{Ok})).
\]

What happens if we break the protocol?
Roadmap

> Modelling interaction in FSP
> Safety — synchronizing critical sections
> Locking for atomicity
> The busy-wait mutual exclusion protocol
> **Conditional synchronization**
> Checking Safety properties
Conditional synchronization

A lock *delays* an acquire request if it is already locked:

\[
\text{LOCK} = ( \text{acquire} \rightarrow \text{release} \rightarrow \text{LOCK} ).
\]

Similarly, a one-slot buffer delays a put request if it is full and delays a get request if it is empty:

\[
\text{const N} = 2
\]
\[
\text{Slot} = ( \text{put}[v:0..N] \\
\rightarrow \text{get}[v] \\
\rightarrow \text{Slot} ).
\]
Producer/Consumer composition


Consumer = ( get[x:0..N] -> Consumer ).

||Chain = ( Producer || Slot || Consumer )
Wait and notify

A Java object whose methods are all synchronized behaves like a monitor

Within a synchronized method or block:

- `wait()` suspends the current thread, releasing the lock
- `notify()` wakes up one thread waiting on that object
- `notifyAll()` wakes up all threads waiting on that object

NB: Outside of a synchronized block, `wait()` and `notify()` will raise an `IllegalMonitorStateException`

Always use `notifyAll()` unless you are sure it doesn’t matter which thread you wake up!
class Slot<Value> implements Buffer<Value> {
    private Value slotVal;  // initially null

    public synchronized void put(Value val) {
        while (slotVal != null) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        slotVal = val;
        notifyAll();  // make waiting threads Runnable
        return;
    }

    ...
... public synchronized Value get() {
    Value rval;
    while (slotVal == null) {
        try {
            wait();
        } catch (InterruptedException e) { }
        catch (InterruptedException e) { }
    }
    rval = slotVal;
    slotVal = null;
    notifyAll();
    return rval;
}
Active objects have their own thread.

Producers and Consumers are active objects that communicate and synchronize through a shared buffer.
Active objects

```java
abstract class ActiveObject extends Thread {
    protected int count;
    ActiveObject(String name, int count) {
        super(name);
        this.count = count;
    }
    public void run() {
        int i;
        for (i=1;i<=count;i++) {
            this.action(i);
        }
    }
    protected abstract void action(int n);
}
```

An active object has a thread of its own.
A generic Producer puts count messages to the slot:

```java
abstract class Producer<Value> extends ActiveObject {
    protected Buffer<Value> slot;
    Producer(String name, int count, Buffer<Value> slot) {
        super(name, count);
        this.slot = slot;
    }
    protected void action(int n) {
        slot.put(produce(n));
    }
    protected abstract Value produce(int n);
}
```
... and the Consumer gets them:

```java
abstract class Consumer<Value> extends ActiveObject {
    protected Buffer<Value> slot;
    Consumer(String name, int count, Buffer<Value> slot) {
        super(name, count);
        this.slot = slot;
    }
    protected void action(int n) {
        consume(n, slot.get());
    }
    protected abstract void consume(int n, Value val);
}
```
public class ProducerConsumerDemo {
    ...
    private class FruitProducer extends Producer<String> {
        protected String wares;
        FruitProducer(...) { ... }
        protected String produce(int n) {
            String message;
            message = wares + "(" + n + ")";
            System.out.println(getName() + " put " + message);
            return message;
        }
    }
    private class FruitConsumer extends Consumer<String> {
        ...
    }
}
Multiple producers and consumers may share the buffer:

```java
public class ProducerConsumerDemo {
    static int COUNT = 5;
    ...
    public void demo() {
        Buffer<String> slot = new Slot<String>();

        new FruitProducer("Peter", COUNT, slot, "apple").start();
        new FruitProducer("Paula", COUNT, slot, "orange").start();
        new FruitProducer("Patricia", COUNT, slot, "banana").start();

        new FruitConsumer("Carla", COUNT, slot).start();
        new FruitConsumer("Cris", 2*COUNT, slot).start();
    }
    ...
}
```
Roadmap

> Modelling interaction in FSP
> Safety — synchronizing critical sections
> Locking for atomicity
> The busy-wait mutual exclusion protocol
> Conditional synchronization
> **Checking Safety properties**
A **safety property** asserts that nothing bad happens

Use ERROR process (-1) to detect erroneous behaviour

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

Trace to property violation in ACTUATOR:

command

command
Safety — property specification

**ERROR** conditions state what is not required

In complex systems, it is usually better to specify directly what *is* required.

**property** SAFE_ACTUATOR

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

Trace to property violation in SAFE_ACTUATOR:
respond
Checking Busy-Wait (revisited)

**property** \( \text{Ok} = ( a \rightarrow b \rightarrow \text{Ok} \\
| c \rightarrow d \rightarrow \text{Ok} ). \)

Ok is a *deterministic process* that specifies which traces are safe — everything else is in ERROR.

**Contrast:**

\( \text{Ok} = ( a \rightarrow ( c \rightarrow \text{ERROR} \ |
| b \rightarrow \text{Ok} ) \\
| c \rightarrow ( a \rightarrow \text{ERROR} \ |
| d \rightarrow \text{Ok } ) ). \)
A **safety property** \( P \) defines a *deterministic process* that asserts that any trace including actions in the alphabet of \( P \) is accepted by \( P \).

**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 behaviour.

> If a behaviour can occur which violates the safety property, then ERROR is reachable.
Why must properties be deterministic to be transparent?

Consider:

\[ \text{property } P = (a \rightarrow b \rightarrow P | a \rightarrow c \rightarrow P). \]

Is \(a \rightarrow b\) allowed or not?
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 behaviours in its alphabet.
What you should know!

- How do you model interaction with FSP?
- What is a critical section? What is critical about it?
- Why don’t sequential programs need synchronization?
- How do locks address safety problems?
- What primitives do you need to implement the busy-wait mutex protocol?
- How can you use FSP to check for safety violations?
- What happens if you call wait or notify outside a synchronized method or block?
- When is it safe to use notifyAll()?
- What are safety properties? How are they modelled in FSP?
Can you answer these questions?

- What is an example of an invariant that might be violated by interfering, concurrent threads?
- What constitute atomic actions in Java?
- Can you ensure safety in concurrent programs without using locks?
- When should you use synchronize(this) rather than synchronize(someObject)?
- Is the busy-wait mutex protocol fair? Deadlock-free?
- How would you implement a Lock class in Java?
- Why is the Java Slot class so much more complex than the FSP Slot specification?
- How would you manually check a safety property?
- Why must safety properties be deterministic to be transparent?
License

You are free:
• to copy, distribute, display, and perform the work
• to make derivative works
• to make commercial use of the work

Under the following conditions:

**BY:** Attribution. You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

• For any reuse or distribution, you must make clear to others the license terms of this work.
• Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

http://creativecommons.org/licenses/by-sa/2.5/