Synchronisation and Restriction in CCS, flowgraphs, and Synchronized in Java

Model-based Design of Distributed and Mobile Systems Lecture 3: Spring 2005

Mikkel Bundgaard
mikkelbu -at- itu.dk

Department of Theoretical Computer Science
IT University of Copenhagen
Overview of the Lecture

The structure of the lecture:

- Selected Exercises of the Last Lecture
- Synchronisation and Restriction in CCS
  - Flowgraphs and Connectivity
- Java Threads
- Interference
- Synchronisation in Java
- Recursive Locking
- Java 1.5 java.util.concurrent
- Reminder on Mandatory Exercise
- Exercises
Exercises of the Last Lecture

- The solution guide for some of the questions is available from the course homepage.

- Solution to exercise on slide 29
  - Given that all the possible traces (to a terminal state) of a process is the following: \( x, y \) and \( x, z \)
  - Does this uniquely define the LTS (hint: non-determinism)?
    - If yes, then draw the LTS.
    - Otherwise, why not? (give a counter-example with two LTSs containing these traces).

- Find the possible initial transitions of the following process
  \[ \text{trans}1.0 + (\text{trans}2.0 \mid \text{trans}3.0) \]
  using the transition semantics.

- Check the result in \texttt{MWB} with the command \texttt{transitions agent},
  which prints out all transitions of \texttt{agent}.?
Exercises of the Last Lecture — Continued

\[
\text{Act: } \alpha.P \xrightarrow{\alpha} P
\]

\[
\text{Sum: } \frac{P_j \xrightarrow{\alpha} P'_j}{\sum_{i \in I} P_i \xrightarrow{\alpha} P'_j} \quad (j \in I)
\]

\[
\text{Com1: } \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}
\]

\[
\text{Com2: } \frac{P \xrightarrow{\alpha} P'}{Q \mid P \xrightarrow{\alpha} Q \mid P'}
\]

\[
\text{Con: } \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad (A \overset{\text{def}}{=} P)
\]

\[
\text{Sync: } \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}
\]

\[
\text{Res: } \frac{P \xrightarrow{\alpha} P'}{(^L)P \xrightarrow{\alpha} (^L)P'} \quad (\alpha, \overline{\alpha} \not\in L)
\]
Exercises of the Last Lecture — Continued

Checking the result in MWB

MWB>agent P(trans1,trans2,trans3) = trans1.0 + (trans2.0|trans3.0)
MWB>transitions P<trans1,trans2,trans3>
Commitments:
|>trans1.0
|>trans2.trans3.0
|>trans3.trans2.0
Synchronisation

- Many process calculi use synchronisation as an underlying mechanism for communication.
- An action and a matching co-action forms together a joined action (a synchronisation).
- The communication is atomic.
- In CCS no information is transferred in the synchronisation, whereas we in the \( \pi \)-calculus transfer names.
Synchronisation in CCS

- Without the rule Sync processes are not able to communicate (they can only run in parallel)
- With the rule SYNC, which models a synchronisation between two parallel processes, processes can communicate

\[
\text{SYNC} \quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}
\]

- \(\tau\) is a special and unique internal action, which have no co-action and which cannot be restricted
- Given the action \(\alpha\) we call \(\overline{\alpha}\) the co-action (and conversely \(\alpha = \overline{\overline{\alpha}}\))
- The input action on \(a\) is the co-action to the output action on \(\overline{a}\) and vice versa
Example on synchronisation

- A simple example

```
MWB>agent Simple(a) = a.0 | 'a.0
MWB>transitions Simple<a>
Commitments:
|>t.0              (*)
|>a.'a.0
|>'a.a.0
MWB>
```

- 3 possible actions \( \tau, a, \) and \( \overline{a}, \) since we can use the rules SYNC, COM1, and COM2

- Notice from the example above that synchronisation is not per se forced in CCS, a normal action is just as possible
  - With respect to the actions \( \overline{a} \) and \( a \) we can say that the environment (or us in the MWB) provides the missing co-action.
Name Equality in MWB

Consider the following process

\[ \text{MWB}\text{\textgreater agent Names}(a,b) = a.0 \mid 'b.0 \]

which takes two arguments and inputs on one and outputs on the other in parallel

\[ \text{MWB}\text{\textgreater transitions Names}^{<a,b>} \]
Commitments:
\[ [a=b] > t.0 \quad (\ast) \]
\[ > a. 'b.0 \]
\[ > 'b. a.0 \]

Due to the implementation of MWB (it is a tool for the \( \pi \)-calculus) it always ask if two different names are equal

these transitions should never be chosen (unless you know exactly what you are doing ;-) )
Restriction

- Restriction can be thought of as a confinement of the scope of an action
- Restriction can control which parties that communicate

\[
\text{RES} \quad \frac{P \xrightarrow{\alpha} P'}{(\sim L)P \xrightarrow{\alpha} (\sim L)P'} (\alpha, \bar{\alpha} \notin L)
\]

- Since \( L \) is a set of names we cannot restrict the internal action \( \tau \)
- From outside of the restriction it is not possible to synchronise with an action in \( P \) which is in the set \( L \)
Examples on Restriction

The following expression

\[
\text{MWB}\text{>agent Res(a) = a.0 | (} \sim a \text{) (}'a.0) \\
\text{MWB}\text{>transitions Res<a>}
\]
Commitments:
\[
|a.(\sim a_2)'a_2.0
\]

can only do a \text{a} transition and no synchronisation. \textbf{Why ?}

whereas the following expression

\[
\text{MWB}\text{>agent Res2 = (} \sim a \text{)(a.0|}'a.0) \\
\text{MWB}\text{>transitions Res2}
\]
Commitments:
\[
|t.0
\]

can only do a synchronisation and neither an \text{a} nor an \text{"a} transition. \textbf{Why ?}

So this way we can \textbf{control} which parties that communicates

Later the restriction operator will play a much more powerful role
Solution to Name Equality

Recall the process

\[ \text{MWB} > \text{agent Names}(a, b) = a.0 \mid \, 'b.0 \]

and the problem

\[ \text{MWB} > \text{transitions Names}<a,b> \]
Commitments:
\[ [a=b] > t.0 \quad (*) \]
\[ > a.'b.0 \]
\[ > 'b.a.0 \]

A solution is to use name restriction to force MWB to consider the different

\[ \text{MWB} > \text{agent Names} = (^a,b)(a.0 \mid \, 'b.0) \]
\[ \text{MWB} > \text{transitions Names} \]
NO commitments.
Non-determinism

Before we only had non-determinism in:
- **Sum** (eg. several choices with the same action \(a.P + a.Q\))
- **Scheduling** (eg. different interleavings of \(a.b.0 | c.d.0\))

Now we also have non-determinism in **synchronisation**

MWB\texttt{>agent NonDeterminism}(a) = a.0 | 'a.0 | 'a.0
MWB\texttt{>transitions NonDeterminism}<a>
Commitments:
\[\begin{align*}
|&t.'a.0 & (*) \\
|&t.'a.0 & (*) \\
|&a.'(a.0 | 'a.0) \\
|>'a.(a.0 | 'a.0) \\
|>'a.(a.0 | 'a.0)
\end{align*}\]

So the two outputs on 'a compete for the single input a
Flowgraphs or Structure Diagrams

Flowgraphs describe the connectivity of the system (along which names a process can communicate)
- User can input on write and output on in
- Cop can input on in and output on out

Static connectivity in CCS (dynamic in π-calculus)

Restriction changes the flowgraph

Java Threads

- A **thread** is a single (and separate) sequential flow of control within a program.
- Threads run **independently** and **concurrently** of each other.
  - In practice, on a single-processor machine, they are scheduled in turns, and thus run **interleaved**.
- One can use threads to **separate tasks**.
- **Examples:**
  - handle calls to a webservice by separate threads
  - use a separate thread to wait for I/O (e.g. from keyboard or network)
Java Threads — Creating a Thread

- The `Thread` class implements a thread with an empty `run()` method, and thus by default does nothing.

- You may provide a run method by:
  - Subclassing `Thread` and override `run()`
  - Call the `Thread` constructor with an object implementing the `Runnable` Interface

- A thread can be in three states:
  - Created, Alive, and Dead (i.e. terminated)

- The Alive state can be subdivided in:
  - Running, Runnable, and Non-runnable

- The transitions depend on scheduling and Thread priorities
Java Threads — A thread as a state machine

- The actions **start** and **stop** (deprecated) are actions of the Thread API that respectively brings the thread alive and terminates it.

- **Note:** We have abstracted away from several actions of the API as well as internal states.

- The actions **(runs)** and **(ends)** are internal actions, representing respectively that the run method runs and that it ends.
Java Threads — The alive state refined

The actions (dispatched) and (getnotified) represent respectively that the thread is dispatched by the scheduler or notified.

 Alive

The actions 'yield, 'sleep, and 'wait are called by the thread itself.

Note: that 'wait is not part of the Thread API and will be called from within another objects method, called by the thread.
Interference

- We call updates that are lost for **destructive updates**.
- Destructive update, caused by the arbitrary interleaving of read and write actions, is termed **interference**.
- **Standard solution** to interference:
  - **Critical regions** must only be executed by one thread at a time.
  - So
    - **Mutually exclusive** access to the methods that access the object.
  - Mutual exclusion can be modeled as atomic actions in CCS.
Example where it goes Wrong in Java

- Problems can arise, when several threads can modify the same data (shared data)
- Consider the following method `increment`

```java
void increment () {
    int temp = value; // read value
    Simulate.HWInterrupt (); // a hardware interrupt
    value = temp + 1; // write value
}
```

- If two threads calls `increment` several times, updates can be lost
Mutual Exclusion in Java

- Java provides the keyword `synchronized` to make methods or blocks mutually exclusive.
- A low-level (simple) form of handling the problems arising from concurrency, but sufficient for the most cases.
- Synchronized also synchronises the whole of thread memory with "main" memory (only dirty variables and not relevant for most people).
- In Java 1.5 a new package for concurrency `java.util.concurrent` (see later slides).
Implementation of Synchronisation

- All objects and classes have a lock
- At most one Thread at a time can “own” the lock of an object or class
- A thread has to acquire the lock (either from the object or the class) before entering a synchronized method (or block)
- When the thread leaves the method (or block), whether it completes natural or by throwing an exception, the lock is released
- Synchronisation is block-based (lexical scope)
- Based on the monitor model explained in the next lecture
public void synchronized method () {...}

- If an object contains several synchronized methods only one of them can be active at a time
- This does not affect the non-synchronized methods
- Making increment synchronized solves the problem

synchronized void increment () {...}

- Only one thread in the method increment at a time
synchronized (object) { statements }

- Access to an object may also be made mutually exclusive by using the synchronized statement

- We could also solve the problem with increment by using an external lock by both threads (an alternative, but less elegant solution)

synchronized (lock) \{ obj.increment(); \}

- Why is this solution less elegant?
A class is called **thread safe** if it is protected against multiple and concurrent access’ from several threads (and this is implemented correctly)

Thread safety in the Java class hierarchy:
- There is a tendency in the Java API that less methods are synchronized (**Vector** vs. **ArrayList**)
- Synchronisation can then be achieved from the “outside” or using the following template

```java
List list = Collections.synchronizedList(new ArrayList(...));
```
synchronized void foo() {
    count += 2;
}

is equivalent to

void foo() {
    synchronized(this) {
        count += 2;
    }
}
Equivalent Synchronisations Per-class

```java
synchronized static void eggs() {
    count += 2;
}
```

is equivalent to

```java
static void eggs() {
    synchronized(S.class) {
        count += 2;
    }
}
```

Where the method `eggs` is contained in class `S`

- (or just `synchronized(getClass())`)
Recursive Locking

```java
public synchronized void increment(int n) {
    if (n > 0) {
        ++value;
        increment(n-1);
    } else return;
}
```

- Once a thread has acquired the lock of an object it can call other synchronized methods on that object without having to wait to acquire the lock again.
- Otherwise the method above would deadlock (why?)
- The lock counts how many times it has been acquired (why?)
- Sometimes also called Reentrant locks
Problems with Performance

- **Performance penalties** on several levels:
  - **Overhead** of the synchronisation implementation, e.g. acquiring/releasing a lock
  - **Less overhead** in newer implementations of JVM
  - Experimental versions of JVM can remove many of the unnecessary synchronized annotations

- The possibility of introducing **deadlock** and **liveness** problems

- Problems between multiple computations that do not require mutual exclusion
  - Consider an object containing two distinct datastructures and these datastructures do not depend on each other

- **Possible solution**?
Concurrency in Java 1.5

- Java 1.5 contains a new package `java.util.concurrent`
- Inspired from Doug Lea’s `util.concurrent` package
- Thread safe queues, Timers, locks (including atomic ones) and other synchronisation primitives, a thread task framework, …
- `FutureTask` (a cancellable asynchronous computation)
- `SynchronousQueue` (blocking each put must wait for a take, and vice versa)
- `Semaphore` (to be used instead of `wait()` much more flexible)
- `ConcurrentHashMap` (A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates)
Concurrent Programming in Java 1.5 Cont. — Executor

- Fork a background thread to execute a task by simply creating a new thread for the task:

  ```java
  new Thread(new Runnable() { ... }).start();
  ```

- Possible overhead creating/deleting threads
- How to effectively schedule tasks,
- **Solution:** Thread pools
- In Java 1.5 the interface `Executor` defines a single method `void execute(Runnable command)`, that executes the given `command` at some time in the future

  - **Decoupling** task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc.

  - The command can be executed in a new thread, in a pooled thread, or in the calling thread, depending on the implementation

- **Subclasses:** `ScheduledThreadPoolExecutor`, `ThreadPoolExecutor`
Concurrence in Java 1.5 Cont. — Future

- Sometimes you want to start a process **asynchronously**, hoping that the results of that process will be available when you need it later.
- The interface **Future** provides the means for this.
- Methods to check if the computation is **complete**, to **wait** for its completion, and **cancellation**

**Subclass:** **FutureTask** is a cancellable asynchronous computation

```java
FutureTask futureImage = new FutureTask<>();
Runnable command = futureImage.setter(new Callable() {
    public Object call() { return renderer.render(rawImage); }
});

executor.execute(command); // start the rendering process
// do other things while executing

drawImage((Image)(futureImage.get())); // use future
```
Futures in Other Languages

- Futures have also been added to a variant of the C# language
- This variant is based on the Join calculus (a fundamental calculi like the π-calculus)
- Objects have both synchronous and asynchronous methods
- A class defines a collection of chords

```csharp
class Buffer {
    String get() & async put(String s) {
        return s;
    }
}
```
Mandatory Exercise 1

- **What**: Hand in mandatory exercise 1
- **Where**: in my (Mikkel Bundgaard) pigeon hole in the theory department in the 4C Corner of the building
- **When**: the 23/2-2005 no later than 12.00
Exercises — Train Crossing

- **Implement** the “Train/Car crossing” example (from the article “Lecture notes about processes”) in **MWB**
- Try **stepping** through the code (do not use the transitions, where **MWB** asks if two different names are equal [a=b])
- **Argue** why it is not possible for both the car and the train to cross at the same time
- **Draw** the transition graph (LTS) for each of the components: **Road**, **Rail**, and **Signal**
- (only for the brave) draw the transition graph (LTS) for **Road | Rail | Signal** and **compare** it with that for **Crossing**
Exercises — Java Threads

This exercise is just to understand the two possibilities for creating threads (extending Thread and implementing Runnable), and should be straightforward (almost not an exercise)

- Download the source code Creating Threads from the lecture plan and understand the source code
- Compile the source code and execute the files TwoRunnableTest and TwoThreadsTest
- Try experimenting with the time the threads sleep
- Why does sleep throw the exception InterruptedException?
Exercises — Java Synchronized

Download the source code for Counter from the lecture plan (under Java Synchronisation Exercise)

Compile and run the code and explain why it goes wrong

Try solving the problem using these different tactics:

- synchronize the increment method,
- synchronize the code-block inside the increment method, and
- make the threads synchronize on some shared lock