COMP2310/COMP6310: Concurrent and Distributed Systems

Semester 2 2014

Assignment 1

A Bakery Simulation

Deadline: 17:00 on Friday 05 September

(Please report any errors, omissions and ambiguities to the course lecturer)
(Amendments to this document after its release in non-draft will be marked in blue)

This assignment is worth 18% of your total course mark. It will be marked out of 36.

Submission

This assignment must be submitted electronically. This can be done using the following commands (according to your enrollment):

```
submit comp2310 ass1 ReadMe.txt Bakery.lts Bakery.java [*.java]
submit comp6310 ass1 ReadMe.txt Bakery.lts Bakery.java [*.java]
```

Extensions and Late Penalties

See the course Assessment page. Note: late penalties apply even if you submit just one file after the deadline.

Objectives

- To model a concurrent system of moderate size, and its safety and liveness properties.
- To implement the model using threads with monitors.

The Bakery

A bakery with *NS* servers (shop attendants) can service *NC* customers. It serves buns, with at most *NB* buns being served in a single transaction. It can do this fairly by dispensing tickets (numbered 0,1, 2,...) in the order of their number. An incoming customer (with customer number *c*) **take**s a ticket, then the next available server (with the server number *s*) **call**s the number t of next ticket to be served. Customer *c* then **pay**s server *s* for *b* buns, presenting the ticket *t* as s/he does so. Server *s* then gives the *b* **bun**s to customer *c*.

For the ticket dispensing part of the system, it can be noted that ticket numbers can be reused after every *NC* tickets are dispensed. For example if *NC=3* and the following sequence of tickets:

```
0 1 2 0 1 2 0 1 2 0
```

was dispensed so far, with two green tickets taken but not yet called, ticket `2` would be the next to be called, and ticket `0` to be called after that.

System Description

Simple system. This is for the simple case of *NC=2, NS=1, NB* ≤ *2*. Ticket reuse will be necessary for the modelling part of the assignment (to keep the number of states manageable), but is optional for the implementation.

Intermediate system (part 1). The model (FSP) should be able to handle the case *NC* ≤ *3, NS* ≤ *2, NB* ≤ *2* (at least), and the implementation should have no particular restrictions on these parameters (note: for testing this level in FSP, keep *NB=1*, to keep the number of actions manageable).

Intermediate system (part 2). The above scenario assumes the servers have an infinite supply of buns to give to customers. We will extend the system so that a server at any time has *0* ≤ *b* ≤ *NB* buns (initially *b = NB*); if a customer **pay**s for *b'* > *b* buns, the server must wait for a **topup** from a cook (who has an infinite supply). After this, the server has *NB* buns, and can proceed with the giving of *b'* **bun**s (note: for testing in FSP, keep *NC* ≤ 2). It is optional whether the **topup**s occur only at the time of giving buns or at any time.
The Bakery management are becoming concerned that some of the customers might be misusing the ticket dispensing part of the system, i.e. taking more than 1 ticket before paying, or presenting a ticket which they didn't take. Specify an FSP `property` that asserts that the customers are not doing that, and a composition with that property that verifies this (and thus reassure the management!). In your implementation, the (bakery) monitor should also check this (and other kinds of correct customer behavior).

Extended System. As well as being temperamental, cooks are notorious for being unfair! Our bakery's cook is no exception, being quite capable of starving some of the servers (not to mention their unfortunate customers!). Specify FSP *progress properties* that each server will eventually get a **topup**, and write an FSP composition that shows that (using *action priority*) that this system does indeed violate progress. Extend the system to provide a solution to this problem. Show that this system does not violate the progress property.

Requirements

You submission must include the following:

- The plain text file called `ReadMe.txt`, created from the following template file, with the requested information filled in, including a disclaimer. Note that significant contributions may require a revision of your final mark, as this is intended to be an individual assignment.

- an FSP file called `Bakery.lts` containing a description of your model. The model should be free from deadlock and (without action priority) be able to make progress. You make find the hints for large(ish) FSP programs helpful.

- a file called `Bakery.java`; this contains a definition of the `Bakery` class, which has the `main()` method to start the system. Classes for the components of your system may be put in this file or in files separately submitted.

After the command `javac Bakery.java` (clarifiaction: it must compile correctly from the same directory as where the file is, i.e. *not* be embedded in a sub-directory within some package), the command:

```
    java -ea Bakery [-c] [-f] [NC] [NS] [NB]
```

should run the simulation with *NC* customers (default value 2) and *NS* servers (default value 1) and *NB* as the bun limit (default value 2). If `-c` was specified, the system has a cook topping up the buns; if `-f` was specified, the cook should top-up the servers in some fair fashion.

`main()` must create a object from the provided `BakeryParam` class, which will obtain the simulation's parameters from the command line and shell environment variables. It must similarly create an object from the provided `BakeryEvent` class.

Each customer must be implemented by a separate thread (more threads can be optionally used for other parts of the system).

When your code completes an *action* (from your FSP model), it must signify this by calling the corresponding event logging method in the provided `BakeryEvent` class. The `sleepEvents()` method should be used for sleep of random intervals after each of these. To achieve both variability and reproducibility with randomly generated values, you should either use the `BakeryParam`'s `getRandVal()` method or its random number seed directly to seed your own generator.

Your implementation should follow the General Guidelines for Writing Monitors

Lines in all 3 files should be formatted to a standard width of 80 characters. Your model and implementation files should follow standard programming style conventions. In particular neat and consistent code layout and adequate commenting in both are important.

Your implementation may attempt any of the three levels of the system; please only submit for marking the code for the highest level. Marks will be allocated to the system levels approximately as follows:

System	**COMP2310**	**COMP6310**
Simple	60	50
Intermediate I	80	70
Intermediate II	100	90
Extended	100	100

The model and implementation code do not have to be at the same level (in which case this should be explained in `ReadMe.txt`); there will be approximate weighting of 1/3 for the model and 2/3 for the implementation (except for the Extended level, where they will be 50% each). A model/implementation at a particular level with serious flaws/bugs is unlikely to score more marks for code attempted at a higher level.

Bakery Parameter and Event Classes

The `BakeryParam` class will provide some ways of controlling the simulation. You can adjust these values in the shell which you run your program from, e.g. for the Bash shell:

```
export BakerySeed=50; java -ea Bakery
```

This will enable repeatability in random number generation between runs, which may be helpful when debugging. To return to default behavior:

```
    unset BakerySeed
```

in which case, the random seed will be time dependent. Other environment variables used by this class are `BakeryMaxEvent` (this will cause termination after this number of events are recorded) and `BakerySleepMax` (which sets the maximum sleep time in `BakeryEvent.sleepEvents()`), and `BakeryNoGui` (suppresses the Bakery GUI when set).

The interface for the `BakeryParam` class is:

```
public BakeryParam(String[] args); // args as from main(String[] args)
int  getNC(); int getNS(); int getNB();  // returns corresponding parameter
bool useCook();      // returns if a cook tops up the buns
bool fairCook();     // returns if a cook tops up with some fairness policy
long getRandSeed();  // returns a random seed for the game
int  getRandVal(int max); // returns a random number in 0..max-1
```

The `interface' for the class `BakeryEvent` class is as follows:

```
BakeryEvent(BakeryParam bakP);
// log respective actions + their parameters in a consistent format
void logTake(int custId, int ticketNum);
void logCall(int serverId, int ticketNum);
void logPay(int custId, int serverId, int ticketNum, int numBuns);
void logBun(int custId, int serverId, int numBuns);
void logTopup(int serverId);
void logOtherEvent(String eventDescription);
void sleepEvents(); // sleep for some random time in 0..SleepMax-1 ms
```

For working externally, you can get a zip file for such auxiliary class files and other files needed for the system. It also contains a readme file, which contains instructions for using the GUI and for working externally. On the CS student system, an up-to-date version of the compiled files will be available through your `CLASSPATH` environment and you will not need the zip file there.

Implementation in Languages Other then Java

This may be difficult but is not out of the question - see this file if interested.