Universiti Malaysia PAHANG

Engineering • Technology • Creativity

FACULTY OF COMPUTER SYSTEMS & SOFTWARE ENGINEERING

GROUP PROJECT DEVELOPMENT

COURSE	**:**	**Formal methods**
COURSE CODE	**:**	**BCS2213**
LECTURER	**:**	**Prof. Dr. Vitaliy Mezhuyev**
SUBMISSION DATE	**:**	**16, 17, 18 December 2014**
SESSION/SEMESTER	**:**	**Semester I 2014/2015**
PROGRAMME CODE	**:**	**BCS**

INSTRUCTIONS

1. The **maximum** amount of marks for successful development of the project is **10%**.

2. This mark also includes evaluation of your **group work (5%)**.

3. In this document **10** tasks are specified, so **10 groups should be formed** (containing 5-7 students).

4. Forming groups is your responsibility (and a part of the project assignment). Please form the groups in order they have equal chances, e.g. containing both strong and not very strong students.

5. Each task gives a brief description of the system you need model with TLA and UPPAAL. The result of the project is a TLA specification and UPPAAL time automata for the chosen system.

6. The given tasks are the classical problems of computer science, so you can find a lot of information in internet.

7. TLA and UPPAAL specifications should contain clear, but not superfluous comments.

8. Project report to be given in the WORD file. Here you need clear formulate, what you are really have modelled and what properties you have checked. Give the analysis of properties of the system with different values of system variables.

   The Report should include next items:
   - Title (containing names of all members of your group)
   - Table of content
   - Introduction (importance of the task, your motivation to solve it)
   - TLA Model and its discussion
   - UPPAAL model and its discussion
   - Conclusion (what do you learn from this project work, how can you improve the models)
   - References

   Please also prepare one per group PowerPoint presentation (every person in the group should have own part in the presentation and need emphasize what is his/her role in the project).

9. Please use for discussions the forum (http://kalam.ump.edu.my/mod/forum/discuss.php?d=577) and the chat (http://kalam.ump.edu.my/mod/chat/view.php?id=6583), opened in Kalam.

10. You may contact me with questions by e-mail: mejuev@ukr.net or in my office in B-1. Preferably, if one person from the group will be responsible for discussions (not to answer the same questions many times).

1. Modelling parallel processes with semaphores

In computer science, particularly in operating systems, a semaphore is an abstract data type that is used for controlling access of multiple processes, to a common resource in a parallel programming environment.

A useful way to think of a semaphore is as a record of how many units of a particular resource are available, coupled with operations to safely adjust that record as units are required or become free, and, if necessary, wait until a unit of the resource becomes available. Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems. Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (locked/unlocked, unavailable/available) are called binary semaphores.

The semaphore concept was invented by Edsger Dijkstra in 1965, and has found widespread use in a variety of operating systems.

Develop model of the next problem.

A library has 10 identical study rooms, intended to be used only by one student at a time. To prevent disputes, students must request a room from the front counter if they wish to make use of a study room. When a student has finished using a room, the student must return to the counter and indicate that one room has become free. If no rooms are free, students wait at the counter until someone relinquishes a room.

The clerk at the front desk does not keep track of which room is occupied or who is using it, nor does he or she know if the room is actually being used, only the number of free rooms available, which she only knows correctly if all of the students actually use their room and return them when they're done. When a student requests a room, the clerk decreases this number. When a student releases a room, the clerk increases this number. Once access to a room is granted, the room can be used for as long as desired, and so it is not possible to book rooms ahead of time.

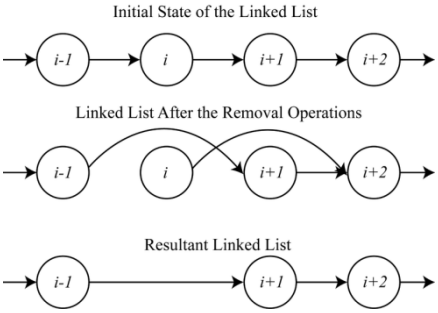
In this scenario the front desk represents a semaphore, the rooms are the resources, and the students represent processes. Lets value of the semaphore in this scenario is initially 10. When a student requests a room he or she is granted access and the value of the semaphore is changed to 9. After the next student comes, it drops to 8, then 7 and so on. If someone requests a room and the resulting value of the semaphore is negative, they are forced to wait. When multiple people are waiting, they will either wait in a queue, or use round-robin scheduling and race back to the counter when someone releases a room.

2. Modelling mutual exclusion mechanism

Develop models of mutual exclusion of access to charged memory in concurrent software system. System has several processes and several resources (here, charged memories).

In computer science, mutual exclusion refers to the requirement of ensuring that no two processes or threads are in their critical section at the same time. A critical section refers to a period of time when the process accesses a shared resource, such as shared memory. The problem of mutual exclusion was first identified and solved by Edsger W. Dijkstra in 1965 in his paper titled: Solution of a problem in concurrent programming control.

For visual description of a problem see Figure below. In the linked list the removal of a node is done by changing the “next” pointer of the preceding node to point to the subsequent node (e.g., if node i is being removed then the “next” pointer of node i-1 will be changed to point to node i+1). In an execution where such a linked list is being shared between multiple processes, two processes may attempt to remove two different nodes simultaneously resulting in the following problem: let nodes i and i+1 be the nodes to be removed; furthermore, let neither of them be the head nor the tail; the next pointer of node i-1 will be changed to point to node i+1 and the next pointer of node i will be changed to point to node i+2. Although both removal operations complete successfully, node i+1 remains in the list since i-1 was made to point to i+1 skipping node i (which was the node that reflected the removal of i+1 by having it's next pointer set to i+2). This problem can be avoided using mutual exclusion to ensure that simultaneous updates to the same part of the list cannot occur.



3. Modelling Producer–consumer problem

The producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. Develop models for the multiple producers and consumers.

4. Modelling first Readers–writers problem

In computer science, the **first and second readers-writers problems** are examples of a common computing problem in concurrency. The two problems deal with situations in which many threads must access the same shared memory at one time, some reading and some writing, with the natural constraint that no process may access the share for reading or writing while another process is in the act of writing to it. (In particular**, it *is* allowed** for two or more readers to access

the share at the same time.) A readers-writer lock is a data structure that solves one or more of the readers-writers problems.

Model the next situation. Suppose we have a shared memory area with the constraints detailed above. It is possible to protect the shared data behind a mutual exclusion *mutex*, in which case no two threads can access the data at the same time. However, this solution is suboptimal, because it is possible that a reader R_1 might have the lock, and then another reader R_2 requests access. It would be foolish for R_2 to wait until $R_1$was done before starting its own read operation; instead, R_2 should start right away. This is the motivation for the **first readers-writers problem**, in which the constraint is added that *no reader shall be kept waiting if the share is currently opened for reading.* (This is also called **readers-preference).**

5. Modelling second readers-writers problem

Model the next situation. Suppose we have a shared memory area protected by a mutex, as was described above (in task 4). This solution is suboptimal, because it is possible that a reader R_1 might have the lock, a writer W be waiting for the lock, and then a reader R_2 requests access. It would be foolish for R_2 to jump in immediately, ahead of W; if that happened often enough, W would starve. Instead, W should start as soon as possible. This is the motivation for the **second readers-writers problem**, in which the constraint is added that *no writer, once added to the queue, shall be kept waiting longer than absolutely necessary.* This is also called **writers-preference**.

6. Modelling the dining philosophers problem

This is an example problem often used in concurrent algorithm design to illustrates synchronization issues and techniques for resolving them. It was originally formulated in 1965 by Edsger Dijkstra as a student exam exercise.

Five silent philosophers sit at a table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers. (An alternative problem formulation uses rice and chopsticks instead of spaghetti and forks.)

Each philosopher must alternately think **or** eat. However, a philosopher can only eat spaghetti when he has both left and right forks (or chopsticks). Each fork can be held by only one philosopher and so a philosopher can use the fork only if it's not being used by another philosopher. After he finishes eating, he needs to put down both forks so they become available to others. A philosopher can grab the fork on his right or the one on his left as they become available, but can't start eating before getting both of them.

Eating is not limited by the amount of spaghetti left: assume an infinite supply. The problem is how to design a behaviour such that each philosopher won't starve; i.e., can forever continue to alternate between eating and thinking assuming that any philosopher cannot know when others may want to eat or think.

In the case of not correct solution the deadlock occur, i.e. a system state in which no progress is possible. To see that designing a proper solution to this problem isn't obvious, consider the following proposal: instruct each philosopher to behave as follows:

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;

- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

This attempt at a solution fails: it allows the system to reach a deadlock state, in which no progress is possible. This is the state in which each philosopher has picked up the fork to the left, waiting for the fork to the right to be put down. With the given instructions, this state can be reached, and when it is reached, the philosophers will ***eternally wait*** for each other to release a fork.

7. Kernel and task interaction.

Develop models of communication of application tasks with a kernel in an operating system on a single processor (SP) computer. There just two types of tasks – kernel and application. System contains one kernel and several application tasks. Each application task has a name and a status variable (active, waiting).

An application task can be in one of two possible states – active or waiting. Only one task in the system is active (running) in the moment of time (due to SP), the rest are waiting. All the tasks registered their names in the waiting list (queue) of a kernel. Kernel cyclically switch the context between tasks, i.e. make them active in the round robin way.

After development of this simple model, add to application task a priority, as an integer with value from 0 to 255, where 0 is highest priority. This will allow to develop model of preemtive scheduling in the concurrent software system. The task with highest priority always becomes active, even in waiting list of a kernel where is other task ready to be active (but having lower priority).

8. Modelling problem about a man, a wolf, a goat and a cabbage

A man once had to travel with a wolf, a goat and a cabbage. He had to take good care of them, since the wolf would like to taste a piece of goat if he would get the chance, while the goat appeared to long for a tasty cabbage. After some traveling, he suddenly stood before a river. This river could only be crossed using the small boat laying nearby at a shore. The boat was only good enough to take himself and one of his loads across the river. The other two subjects/objects he had to leave on their own.

How must the man row across the river back and forth, to take himself as well as his luggage safe to the other side of the river, without having one eating another?

9. Conway's Game of Life

The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970.

The "game" is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves or, for advanced players, by creating patterns with particular properties.

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square *cells*, each of which is in one of two possible states, *alive* or *dead*. Every cell interacts with its eight *neighbours*, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if caused by under-population.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The initial pattern constitutes the *seed* of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a *tick* (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

10. The Cannibals

Three cannibals and three anthropologists have to cross a river.

The boat they have is only big enough for two people. The cannibals will do as requested, even if they are on the other side of the river, with one exception. If at any point in time there are more cannibals on one side of the river than anthropologists, the cannibals will eat them.

What plan can the anthropologists use for crossing the river so they don't get eaten?

Note: One anthropologist cannot control two cannibals on land, nor can one anthropologist on land control two cannibals on the boat if they are all on the same side of the river. This means an anthropologist will not survive being rowed across the river by a cannibal if there is one cannibal on the other side.