Assignment 1: The Unisex Bathroom Problem

The aim of this assignment is:

- to gain familiarity with semaphores
- to gain practice in solving mutual exclusion and synchronization problems
- to gain practice in verification of safety and liveness properties

In order to take fully advantage of help given during the workshop, you should prepare yourself. Before the workshop, you should study the exercise and work out an initial idea for a solution to the problem. In the workshop, help will be available in using Java and Uppaal. Report instructions (Report_structure.pdf) and examination criteria (KritInlupp.pdf) are available on SCIO. Follow the report structure and let the examination criteria guide you in your work. Please observe that assignments reports, which do not follow the given structure or are handed in too late, may be considered as a fail. Moreover, there are some rules (Rules.pdf) concerning plagiarism that apply to any submission.

Download the following files to your home library before you start:

- UB_problem.xml
- UB_problem.q

Suppose there is one bathroom in your department. It can be used by both men and women, but not at the same time.

Solution outline

Please download the Assignment_1 code base for the assignment.

You need to follow the solution outline. You need to add the necessary semaphores to the GlobalState and you need to add the necessary implementation to the classes Women and Man representing the two process types.

You may not add other components and you must use the AndrewsProcess and AndrewsSemaphore components to achieve your solution. In case you are unsure discuss this with the supervisor.

Think also about that you need to do a model verification on the code. Advanced data types or language specific structures can make model verification more difficult. Try to write as clean and straightforward code as possible too as otherwise model verification will be more difficult.

1. Discuss the problem

Define a global invariant and discuss what type of problem this is. It is a variant of one of the classical problems discussed during lectures. Don't skip this step because your solution is supposed to fit the problem.

It is important that when you are defining the problem, you think about the problem you have at hand. A simple software evaluation model you can use is the what, why, how, when and who model with the named aspects. You have a resource allocation problem and you need to think about the following aspects:

- What you need to solve with the resources, also what is your problem
- Why do you need to solve this with respect to the resources, also what could happen otherwise
- How do you solve this with respect to the resources
- “When” do you solve this, which translates in this case to how the ordering in time looks like for the parts of the system / how the interactions looks like in time
- “Who”, also which are the important parts within your system in this case

Think about that in this case, you have functional and extra functional requirements, e.g. for example people need to be able to use the bathroom as a functional requirement and extra functional requirements, like nothing bad will ever happen on an abstract level.

You need to develop three requirements and afterwards you need to consult with the supervisor on the workshop to check if your initial requirements are formulated well enough so that you can carry on with implementing your solution. Please note that afterwards, it can happen that you need to refine your requirements if needed under the implementation process.

2. Develop your first solution in Java and in a timed automata model

From the above outline, develop an implementation in Java and a corresponding timed automata model in XML. Use the file *UB_problem.xml* to model your solution. The xml model contains a template that implements the semaphores. It also contains templates for a man and a woman process. Your task is to extend the templates for the man and woman processes. You should **not** modify the semaphore template.

The Java implementation and the TA model should correspond with each other and model the behavior of the men and the women processes, using **semaphores** for synchronization. **Note:** it's not ok to use any other mechanism for synchronization, such as flags, algorithms or monitors. Allow any number of people to be in the bathroom at the same time but ensure that there never are a woman and a man in the bathroom at the same time. Your solution should avoid deadlock, but it need not ensure liveness in this, your first solution.

2.1. Verify your solution

When the model is finished, you can check its syntax and run the simulator to study its behavior. Maybe you will find that the behavior differs from what you expected in which case you might need to correct your model and hence, the code. When you are satisfied with the simulations you should continue with the verifier. The given queries in *UB_problem.q* may serve as input to the verification. However, the set of provided queries should be considered as a starting point. Hence, **you are expected to define more queries** in order to fully verify the model. Note that you have requirements which you can base your requirements on.

Although model checking is a formal proof, it can only prove properties of your model and there is a risk that the model differs from your implementation. Hence, you still need to test your code. Testing concurrent programs is hard due to the many different behaviors, the low observability and the low controllability. Use write statements to increase observability during testing. For example, it is a good idea to print out the number of women and men in the bathroom whenever anyone enters. Note that you also get a function for this which you can even modify to suit your needs. It might also be a good idea to print out which semaphore that is released and under what circumstances. To increase controllability during testing, you can use *nap statements* to enforce preemptions, you also get the doThings() function to help you with this. Try to place your nap statements in places where you think a preemption will increase the probability to reveal a synchronization fault. Again, you might find it necessary to make corrections as the tests reveals bugs. Just remember to update both the code and the model. **After any modification, you have to run the tests and queries again**.

2.2. Semantic differences

Uppaal is a tool for timed automata and it uses synchronous message passing and shared variables for communication and synchronization. This means that there are semantic differences between your model and your program which have to be accounted for.

- Semaphores: Uppaal does not implement semaphores. Instead there are synchronous message passing. Since this has different semantics than semaphores, we must implement the behavior of semaphores with a template.
  - The **P(s)** operation is replaced by a call **P[s]!** to the semaphore process.
  - The **V(s)** operation is replaced by a call **V[s]!** followed by a transition to a state where the calling process may be blocked.
- Progress: A process may choose to stay in a state forever if nothing forces it to continue. Therefore, we must ensure that if there are allowed transitions, one of them will be executed. We can do this by using an urgent channel *Go*. All transitions, where there is no other synchronization on an urgent channel, must therefore, be labeled with a synchronization on channel *Go*.

3. Second solution: Limit number of people

Copy your code to the appropriate package within the Java project. Modify your solution in the appropriate project so that at most four people are in the bathroom at the same time. The same requirements as in previous task still apply. Again, you must test your solution, you need not however, model your solution in order to formally verify it using the model checker.

There are two reasons for you to save all three solutions:

- It is the first solution that will conform to your model and I will check for conformance as I examine your result. Hence, I need to see this version of the code unless you decide to model the third solution as well (can be hard).
- If you only hand in the last solution (which is the hardest to create) then it must be correct in order to be approved. If I get all three solutions and two of them are correct, I can accept a third solution that is close enough.

4. Third solution: Ensure fairness

Copy the necessary parts from your second solution to the appropriate package and modify it to ensure fairness as well as the requirements for the previous versions. The previous solutions did not require liveness but now you are supposed to present a solution that does not only guarantee liveness but is also fair (rättvis). There might be different interpretations of what it means to be fair. Hence, I advise you to elaborate on that in your report. This part is the trickiest one but you are not required to model this solution in Uppaal. It is strongly advised to consult with the supervisor on a workshop on how you defined fairness and how it affects your requirements.

5. Report your results

Each group member should upload the following files electronically using the uploading tool (Uppgifter) in Scio before deadline. Replace {gId} with the group identity that you have got in Scio. The following files should be uploaded:

- Your report
    - Assignment1_{ gId }.doc (or Assignment1_{ gId }.pdf)
- Your source code files (the whole project, not just the .java files, make sure to test the zip file!)
    - Assignment1_{ gId }.zip
- Your xml model
    - Assignment1_{ gId }.xml
- Your query file
    - Assignment1_ { gId }.q

Your report Assignment1_{ gId }.doc (or Assignment1_{ gId }.pdf) must also be sent to Urkund for analysis (only once per group). The e-mail address is andras.marki.his@analys.urkund.se