Assignment 2: Smokers' problem

The aim of this assignment is:

- to gain familiarity with message passing
- to gain practice in solving mutual exclusion and synchronization problems
- to gain practice in verification of safety and liveness properties
- to gain a deeper understanding 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.doc) and examination criteria (KritInlupp.pdf) are available here. Follow the report structure and let the examination criteria guide you in your work. Please observe that reports, which do not follow the instructions 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:

- Smokers_problem.xml
- Smokers_problem.q

1. Smokers' problem

Assume that you have three smokers, **A**, **B**, and **C**. Together they have an unlimited supply of the ingredients they need to make cigarettes and smoke. The problem is that they only have one ingredient each, **A** has the tobacco, **B** has the paper and **C** has the matches. Moreover, it is not possible for them to give each other anything. They can only fetch missing ingredients from a table. Apart from the three smokers, there is also an agent. The agent repeatedly places two, **randomly chosen**, ingredients on the table. The smoker that has the third ingredient picks up the items from the table and smokes. **As soon as the table is empty**, the agent places two new ingredients on the table. A smoker cannot store anything, which means that the ingredients are left on the table until the smoker that has the third ingredient is ready to pick them up.

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 channels to GlobalState and you need to add the necessary implementation to the classes Agent and Smoker representing the two process types.

You may not add other components and you must use the **AndrewProcess** and **AsynchronousChan<Integer>** components to achieve your solution. *Note that with AsynchronousChan, you are only allowed to use the <Integer> type*. The reason is that you will use the channel only for communication, but not to actually send over resources. In case you are unsure discuss this with the supervisor.

Comment

No global variables are allowed. Hence, the Agent is the only process that can actually access the counters. The idea is that the information concerning the table (i.e., placing and picking up) is communicated between the Agent and the Smokers using *asynchronous message passing*. The counters are however, useful for testing purposes since the Agent can increase them when an item is placed on the table and decrease them when informed that something has been removed. Also, the counters in the smokers can also be used for testing purposes, although note that the counters only can work on local level.

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.

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 smoke 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 SR and a corresponding timed automata model in XML. Use the file *Smokers_problem.xml* to model your solution. The xml model contains a template that implements the channels. It also contains templates for the agent and the smoker processes. Your task is to extend the templates for the agent and smoker processes. You should **not** modify the channel template.

The SR implementation and the TA model should correspond with each other and model the behavior of the agent and the smoker processes, using **asynchronous channels** for synchronization. **Note:** it's not ok to use any other mechanism for synchronization, such as flags, algorithms, monitors or rendezvous. Allow any number of smokers to be smoking at the same time but ensure that only one process can access the table at the same time and that the ingredients are picked up by the right smoker. Your solution should avoid deadlock, but it need not ensure liveness in this, your first solution.

NOTE: please observe that the channel template defined in the xml model constrains your message to one or more integers. Hence, your Java solution must have the same limitation, which is also a reason why you need to use the Integer type for the AsynchronousChan.

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 *Smokers_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. Again, it is a good idea to set up queries related to your previously defined requirements.

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 println statements to increase observability during testing. An example for that is the function *getState()* which you can modify if needed. For example, it is a good idea to print out what's on the table whenever anyone access it. It might also be a good idea to print out which smoker that is picking up what under which circumstances. You can surely come up with more tests. To increase controllability during testing, you can use random naps to enforce preemptions, which you can use *doThings()* for. 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.

- *Channels:* Uppaal does not implement asynchronous message passing. Instead there are synchronous message passing. Hence, we must implement the behavior of asynchronous channels, i.e., the message buffer. An instance of the Channel template is ready to synchronize on send and stores the message until there is a synchronization on receive. Hence, we model the behavior of **receive** and **send** operations using synchronization with the corresponding channel process. Any message is transferred using the global variable msg.
    - **receive** *chanId(msg)* is replaced by *receive[chanId]?* and an assignment *myOutMessage=msg*

    - **send** *chanId(msg)* is replaced by *send[chanId]!* and an assignment *msg=myInMessage*
- *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: Arbitrary number of smokers

Save your Java solution and copy your solution (if you would like to) to the appropriate package. Modify your solution to work with an arbitrary number of smokers. This means that your solution should still work when two smokers are waiting to collect the same type of ingredients. 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.

3. Third solution: Fairness

Discuss why any solution to the Smokers' problem (as defined above) cannot guarantee liveness for a Smoker process while liveness can be guaranteed for the Agent. Save your second Java solution, move the code base even to the third 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). To do this, you need to perform a trade-off between fairness and the given requirements. There might also be different interpretations of what it means to be fair. Hence, I advise you to elaborate on that in your report. It is strongly advised to consult with the supervisor on a workshop on how you defined fairness and how it affects your requirements.

This part is the trickiest one but you are not required to model this solution in Uppaal.

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 Java project, not just the source 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