Lab 4: Modeling and Verification Using UPPAAL

Real Time Systems, Uppsala University, Autumn 2015
Group 2: Pei-Chun Chen, Martin Kjellin, and Jia-Ying Lin

Part 1: Verification Warm-Up

- Can the value of the variable i become strictly greater than 200?
  Query: E<> i > 200
  Answer: no

- Is it true that no matter what happens (always) is the value of i smaller than or equal to 100?
  Query: A[] i <= 100
  Answer: yes

- Is it true that no matter what happens (always) is the value of i strictly smaller than 100?
  Query: A[] i < 100
  Answer: no

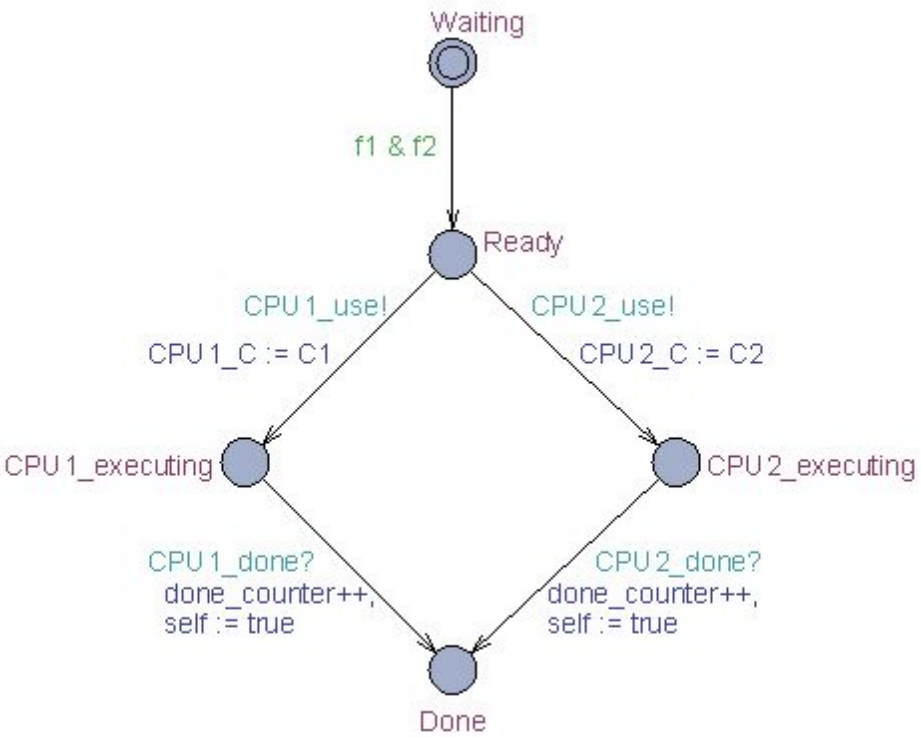
- Is there a run of the system in which i never exceeds 42?
  Query: E[] i <= 42
  Answer: yes

- Is it true that no matter what happens (always) is the value of i greater than or equal to 0?
  Query: A[] i >= 0
  Answer: no

Part 2: Scheduling

In order to express the dependencies between the jobs, we introduced three Boolean parameters (flags): *f1*, *f2*, and *self*. Each job can depend on maximally two other jobs. The parameters *f1* and *f2* indicate if these jobs are finished, while the parameter *self* indicates if the job itself is finished. We also declared one global Boolean variable, initially set to false, for each job (*flagA*, *flagB*, and so on). When the "Job" template is instantiated, the correct Boolean variables are assigned to the parameters. For example, for job *D* in the second problem instance, *f1* is set to *flagA*, *f2* is set to *flagB*, and *self* is set to *flagD*. If there are no dependencies, both *f1* and *f2* are set to true. If there is only one dependency, *f2* is set to true.

As can be seen in the figure below, we check *f1* and *f2* to see if the jobs on which the current job depends are finished. If they are both finished, the current job can enter the *Ready* state and wait to access some idle CPU. It synchronizes with one of the CPUs on the corresponding *use* channel, sets the corresponding *C* variable to the execution time on that CPU, and enters an *executing* state. The job then synchronizes with the CPU on the corresponding *done* channel, increases *done_counter* and sets *self* to true, thereby indicating that it has finished. Finally, the job enters the *Done* state.



In order to generate the trace for the case without dependencies between the jobs, we simply used the query "E<> done_counter == 5", that is, we investigated if there is an execution in which all five jobs are eventually finished. (Yes, they are.)

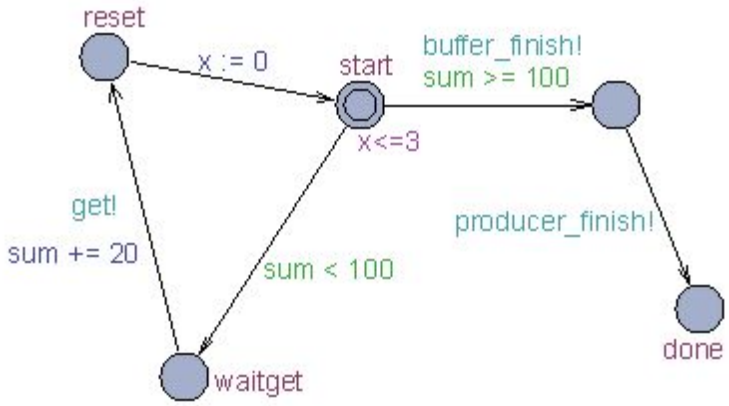
In order to prove the optimality of the given schedule for the case with dependencies between jobs, we used the two queries "E<> done_counter == 5 and y == 8" and "E<> done_counter == 5 and y < 8", that is, we investigated (a) if there is an execution in which all five jobs are finished when the global

clock is 8 and (b) if there is an execution in which all five jobs are finished when the global clock is less than 8. The former query can be satisfied (and the generated trace corresponds to the given schedule), while the latter cannot be satisfied. This proves that the given schedule is actually optimal.

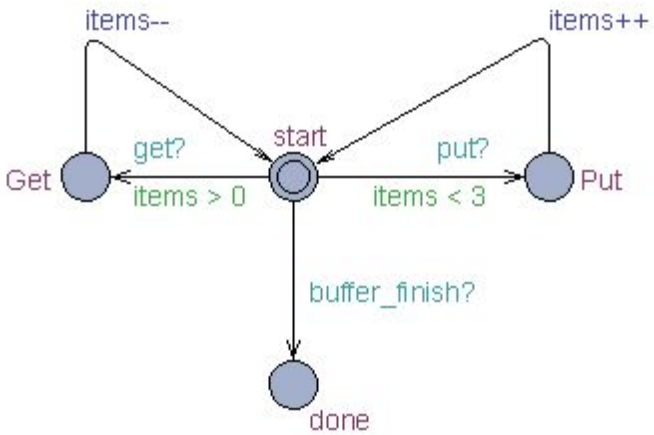
The minimal execution time for the second problem instance is 14. This value can be found using the query “E<> done_counter == 9 and y == t”, where t is initially set to 0 and then incremented until the query is satisfied.

Part 3: A Deadlock

Our Consumer automaton looks like this:



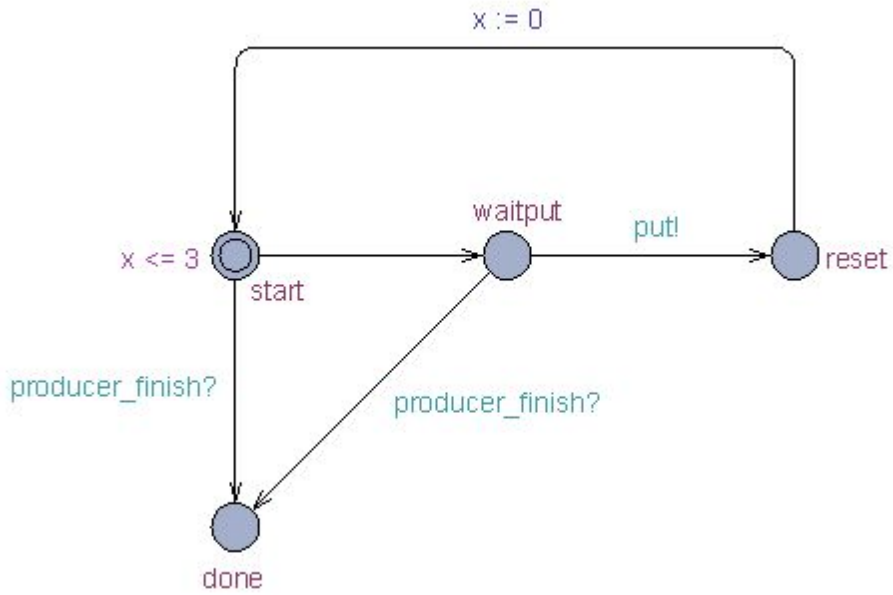
Our Buffer automaton looks like this:



The extended query we used to exclude the "artificial" deadlock was "A[] not deadlock or (Buffer.done and Consumer.done and Producer.done)".

The second deadlock occurs when the Buffer is full, the Producer is in the *waitput* state, and the Consumer tries to terminate the Producer. The Producer is unable to synchronize with the Consumer on the *producer_finish* channel when it is in this state. Furthermore, the Producer is unable to move from this state, since the Consumer will not make any room in the Buffer by getting additional items. Thus, a deadlock has occurred.

We suggest fixing the problem by adding an additional edge between the *waitput* and *done* states in the Producer (see the figure below). Using this edge, the Producer can synchronize with the Consumer on the *producer_finish* channel also when it is in the *waitput* state. This solution seems to fix the problem regardless of the order in which the Consumer terminates the Producer and the Buffer.


x := 0
waitput
put!
x <= 3
start
reset
producer_finish?
producer_finish?
done