Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Networked Systems Group (NSG)

HS 2021

R. Jacob / T. Schneider
based on Prof. L. Thiele's material

Discrete Event Systems

Exercise Sheet 13

This last exercise session puts somehow everything together. You will discover a simulation tool for time Petri nets, which is paramount for the evaluation of your design. You will mainly use the "stepper simulator", which randomly plays the token game. In the last part of the exercise, you will use the embedded model-checker to tune some design parameters.

1 Warming up: Calculating with Petri nets

In this exercise you are supposed to model a function $f_i(x, y)$ using a Petri net. That is, the Petri net must contain two places P_x and P_y that hold x and y tokens respectively in the beginning. Additionally, the net must contain one place P_z which holds $f_i(x, y)$ tokens when the net is dead. The Petri nets are supposed to work for arbitrary numbers of tokens in P_x and P_y.

a) $f_1(x, y) = 5x + y \quad \forall x, y \geq 0$

b) $f_2(x, y) = x - 2y \quad \forall y \geq 0, x \geq 2y$

c) $f_3(x, y) = x \cdot y \quad \forall x, y \geq 0$ Here you may want to use inhibitor arcs. An inhibitor arc between a place and a transition prevents the transition from firing as long as there is at least one token in the place.

Hint Start by creating a net that "duplicate" the number of tokens from P_x in place P_z. Then adapt this net to perform the multiplication.

2 Simulate your Petri nets with TINA

In this exercise, you will learn how to use the modeling tool TINA (**TI**me petri **N**et **A**nalyzer). The objective is to implement your solutions from Exercise 1 and simulate the nets.

a) On our course webpage you will find a simple tutorial for basic uses of the TINA software. Complete the reading of Section 1 before moving on.

b) To practice, create new net files and implement your solutions from Exercise 1. Using the stepper, try them out for different values of x and y.

3 Queue sizing and overflow management of BOLT

BOLT is an ultra-low power processor interconnect that decouples arbitrary application and communication processors with respect to time, power and clock domains. BOLT supports asynchronous message passing with predictable timing characteristics. Therefore, it enables system designers to construct highly-customized platforms that are easier to design, implement, debug, and maintain. You can find more information on BOLT's webpage: www.bolt.ethz.ch

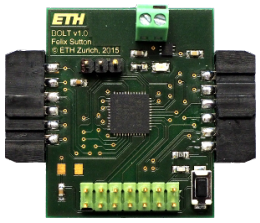


Figure 1: the BOLD processor interconnect

In this exercise, we look at the practical problem of queue sizing and overflow management, using BOLT architecture as a case study. Assume BOLT is connected on one side to a sensor, which delivers data at a given frequency and writes into BOLT queue. At the other side, a communication processor is responsible for reading those messages from BOLT and sending them through a wireless sensor network. The problem is that the network is not always available (other processes also need the bandwidth) and the communication processors have limited memory to store messages before sending them. The BOLT queues can be used as temporary buffers.

The objective is to evaluate the design parameters (queue size, resource management scheme...) such that overflows in the BOLT queue (which would result in packet losses) are avoided.

3.1 The nice and pretty deterministic world

a) This initial model (in the exercise folder) is untimed. Extend it with delays on transitions such that the net follows the earliest firing rule and knowing that:

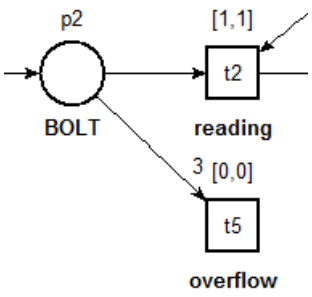
- The sensor produces one message every 5 time units.
- Reading and writing from/to BOLT take one time unit each.
- Sending a message to the network takes one time unit.
- On average, the communication processor uses the network for 10% of the time.

b) Play the token game using the stepper. Is the net bounded? Why is this not surprising?

c) Assume BOLT has now a maximal capacity of two messages. How would modify the net to implement this? Why is this not a "real" solution to our problem?

On the physical system, BOLT does not prevent a write operation if its queue is full, but overwrites old messages. Hence, our previous solution to avoid overflows is not satisfying, as it yields that the write operation is forbidden if there is no more memory available in BOLT, so the systems is actually waiting for the resource to be available.

In our case, the goal is to guarantee there is enough network resource available such that there is no possible overflow in BOLT, without having to control the write operations. As a result, we need our Petri net to model the (possible) overflow of BOLT in order to test our design. Among several options, we choose to do that by adding a sink transition downstream the `BOLT` place, as shown thereafter.



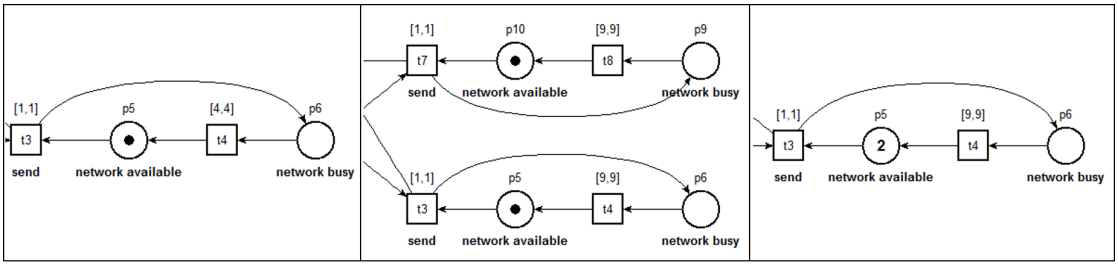
If it happens that 3 tokens are in the `BOLT` place, the overflow transition is enabled and can fire. This is used as a flag to detect an overflow of BOLT.

d) Add the `overflow` place to your model.

e) Let us try to avoid overflows by increasing the amount of `network` resource available. Say, we want to double it on average, such that it matches the generation of sensor data. This can be done in several ways:

- Assume you get the network 20% of the time.
- Assume there are two networks and you get each for 10% of the time. Further assume the communication processor can read and store one message per network (e.g., if there are two networks, you should have 2 tokens in place $p4$). One can either
  - explicitly model two independent networks, or
  - set a second token in the `network available` place.

Those three options are illustrated below



a) b) c)

Try out the three options.

Note 1. You can have multiple nets open simultaneously. Just open multiple GUI in parallel.

Note 2. You can copy/paste parts of a net.

Why solution **c)** does not solve our problem? Which of solution **a)** or **b)** seems the more flexible to you? Why?

f) Assume now that incoming packets arrive in burst of 5 messages every 10 time units. Consider again solutions of type **a)** and **b)** with increased bandwidth:

- One network that you get 50% of the time.
- Five networks that you get 10% of the time each.

Model the bursts and try out these options. Which solution seems the more flexible to you now?

As you can see, even in such basic scenario, the analysis of our system design is not so easy. Understanding of the impact of all design choices is hard. In practice, it is often impossible to take everything into account in one model for real-life systems. This is why test and verification methods have been developed and are common practice nowadays.

3.2 Real-world is non-deterministic

Let us now consider a slightly more realistic model. For each task, there is not one precise execution time anymore but a time interval. For example, for reading and writing from/to BOLT, this represents the Best- and Worst-Case Execution Time (BCET/WCET). The resource management mechanism and the communication processor are also different.

- The processor can read and store one or several messages from BOLT before sending them.
- When the processor accesses the network to send messages, it will keep on sending (i.e., no releasing the resource) until it has no more message left in memory.
- It is uncertain when the network will be available again.

a) Complete the reading of Section 2 of the TINA tutorial.

Our design question is the following:

How much memory does one need in the communication processor in order to guarantee that BOLT never overflows?

To answer this question, we will use the LTL model-checker embedded within TINA. LTL is a different logic than CTL, but for our simple purpose here, we can say the differences are the following

- the E quantifier doesn't exist,
- the A quantifier is always implied,
- G writes $\Box$, F writes $\Diamond$, and X writes $\circ$.

This is summarized in the table below:

CTL	AF a	AG a	EF a	AX a	...
LTL	$\Diamond$ a	$\Box$ a	$\neg \Box \neg$ a	$\circ$ a	...
in TINA	`<> a ;`	`[] a ;`	`- [] - a ;`	`() a ;`	...

b) What does the LTL property $\Diamond t5$ mean in natural language? Does our model verify this property? Are we happy about this?

c) What is the LTL property our system must verify in order to guarantee that BOLT will never overflow?

d) Which place (in the initial state) defines the memory size of the communication processor?

We now want to find the smallest possible memory size of the communication processor in order to satisfy the LTL property above. As already mentioned in the lecture, the usual process for doing this is to find the minimal size by trial-and-error using, e.g., binary search. However, since it is a tedious task (especially with TINA, where the state space analysis needs to be restarted for every new trial), we give the answer: **27**. Instead, let us try to understand why this is the minimal memory size to prevent overflows.

e) Try the property found in **c)** for capacity **26** and **27** with the TINA model-checker. Can you verify that the minimal capacity is really **27**?

f) Have a closer look at the model. Can you figure out how to determine the minimal capacity without using the model-checker?

g) Change the model as follows: Set the interval of transition $t4$ to $[10, 30]$. Now, using your solution from **f)**, infer the new minimal capacity and verify that your solution is correct using the model-checker.