Budapest University of Technology and Economics
Department of Measurement and Information Systems
Fault Tolerant Systems Research Group
Critical Embedded Systems

Formal modelling and verification in UPPAAL

Laboratory

András Vörös
Based on the document written by Dániel Darvas and Gergő Horányi

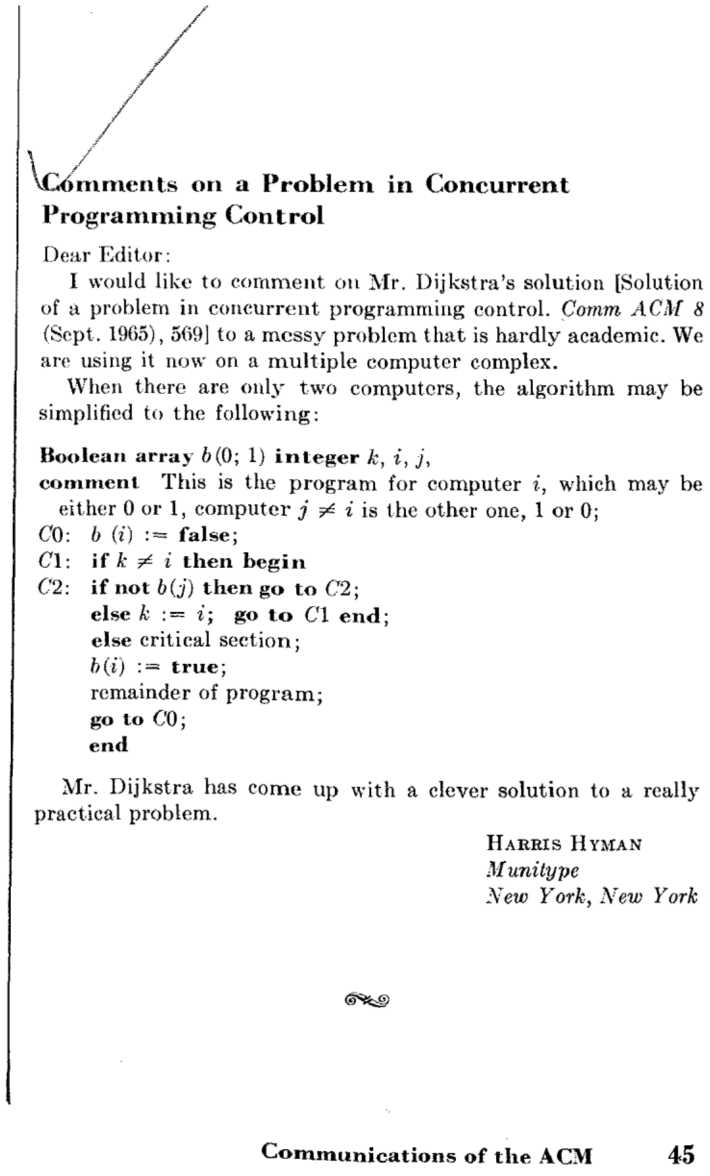
15. October 2013

1 Introduction

In this laboratory we are going to model a mutual exclusion protocol and verify the correctness.

1.1 Mutual exclusion protocol

We are going to model a mutual exclusion protocol, originally published in "*Comments on a problem in concurrent programming control, Communications of the ACM, v.9 n.1, p.45, Jan. 1966*". Figure 1. depicts the original publication.



Comments on a Problem in Concurrent Programming Control

Dear Editor:

I would like to comment on Mr. Dijkstra's solution [Solution of a problem in concurrent programming control. *Comm ACM 8* (Sept. 1965), 569] to a messy problem that is hardly academic. We are using it now on a multiple computer complex.

When there are only two computers, the algorithm may be simplified to the following:

```
Boolean array b(0; 1) integer k, i, j,
comment This is the program for computer i, which may be
  either 0 or 1, computer j ≠ i is the other one, 1 or 0;
C0:  b (i) := false;
C1:  if k ≠ i then begin
C2:  if not b(j) then go to C2;
     else k := i;  go to C1 end;
     else critical section;
     b(i) := true;
     remainder of program;
     go to C0;
     end
```

Mr. Dijkstra has come up with a clever solution to a really practical problem.

HARRIS HYMAN
Munitype
New York, New York

Communications of the ACM **45**

Figure 1. Original publication of the mutual exclusion algorithm

Mutual exclusion refers to the requirement of ensuring that no two processes or threads (henceforth referred to only as processes) are in their critical section at the same time. Here, a critical section refers to a period of time when the process accesses a shared resource.

What do You think, which are the important properties of a mutual exclusion protocol? Can You verify them by inspecting the code above?

In the following let us use a more structured description (which is closer to recent programming languages and easier to read and understand) of the protocol.

This is the working (code) of a participant:

```
turn=0, flag[0]=flag[1]=false;

Protocol (int id) {
      do {
            flag[id] = true ;
            while (turn != id) {
                  while (flag[1-id]) /* do nothing */ ;
                  turn = id;
            }
            CriticalSection(id);
            flag[id] = false;
      } while (true);
}
```

We are going to analyse a system containing 2 participants with `id` = 0 and `id` = 1.

In order to be able to analyse our system, we have to define the properties. What are the requirements (failures) regarding a mutual exclusion protocol?

We are analysing the following properties:

- **Race conditions**: Is it possible that they access the shared resource incorrectly? Is it possible that more than one process access the shared resource simultaneously (more processes enter the critical section simultaneously)?
- **Starvation**: Is it possible that because the incorrect behaviour of the system, a process will not get the resources?
- **Deadlock**: Is it possible that the system gets stuck in a state, and cannot continue its working?
- **Livelock**: Is it possible that the system is working but do not proceed with its tasks? (for example when two processes are calculating who can go to the critical section but none of them enters)

What do You think: do these properties hold for our mutual exclusion protocol?

1.2 Creating the UPPAAL model

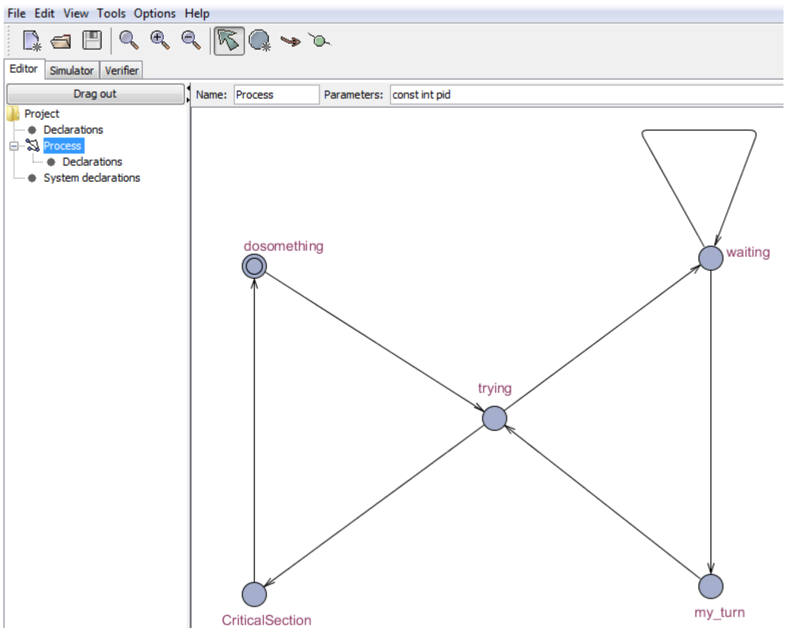
Read the UPPAAL introduction on the homepage of the course!

Design the UPPAAL model of the mutual exclusion protocol! What are the locations in your protocol model? What are the transitions? Can you find a correspondence between the lines of the source code and the locations/edges in the model?

1.2.1 Hint

The following figure (Figure 2.) depicts a possible skeleton of a process. The skeleton can be completed in a way that You associate commands (line of code) to the edges of the skeleton.

The skeleton is parameterized: the parameter of the template (*"Parameters"*) is the pid (the process id).



Name: Process
Parameters: const int pid

Figure 2. Skeleton of the mutual exclusion protocol

After finishing your model, examine the correctness of your model by inspection! Use the simulator of the UPPAAL tool! Does your model behave as it is defined in the code?

2 Verification in UPPAAL

In order to verify our model of the mutual exclusion protocol, we have to overview the verification process. In UPPAAL we can model, simulate and verify our systems. Verification answers the questions we formalize about our system.

UPPAAL supports the so called CTL (Computation Tree Logic) temporal logic as a specification language. With the help of CTL we can express properties and their evolution during the behaviour of the system.

The structure of CTL expressions in UPPAAL is the following: «temporal operator» «logical expression». Logical expressions are conditions of variables or clock variables, for example `money==10000, cl>0`, and also expressions of locations like `p1.CriticalSection`. Temporal operators are the following: `A[], A<>, E[], E<>`. In order to understand the semantics of these operators we have to be familiar with the concept of *computation tree*.

The nodes of computation tree represent states (of the state space) and the edges represent the possible state changes (transitions). The construction of the tree starts from the initial state of the system, let be this the initial node v. The next step is to search for the next states reachable from the initial state in one step, these nodes are labelled by w_i. After it, add all $v \rightarrow w_i$ edges to the graph, and continue this process for every w_i. Note that if a state x is reachable both from states v and w, then there will be two nodes assigned to x, the edges from v and w will not point to the same node! It is also important to note that the depth of this tree is potentially infinite; it is only finite if every branch reaches a deadlock state.

Let now consider a simple traffic signalling system and its states. the initial state is red, when traffic is prohibited. From this state the system goes to red-yellow state, than it lets the traffic pass as it becomes green. Yellow signal follows the green and finally we go into the red state. In addition, from every state the signalling system can go to blinking yellow state. From this blinking yellow state the signalling system can only go to red state: this way it has a safe behaviour. First part of the computation tree of this signalling system is depicted on Figure 3.

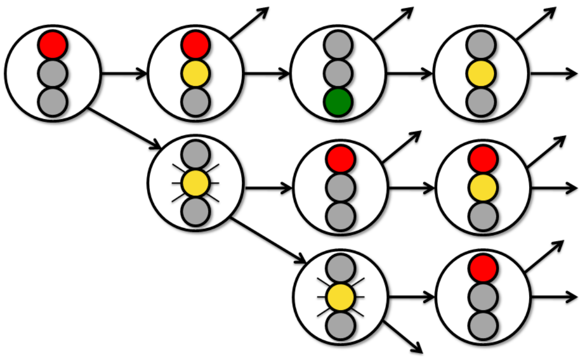


Figure 3. Part of the computation tree of traffic signalling system

We are able to define the semantics (meaning) of the different temporal operators of UPPAAL:

- A[] « logical expression »: true, if the expression is true for every node in the computation tree.
- A<> « logical expression »: true, if there is a state in every branch from initial state, where the expression is true.
- E[] « logical expression »: true, if the expression is true in the whole trace starting from a branch from the initial state.
- E<> « logical expression »: true, if at least one sequence from at least one branch which contains a state where the expression is true.

The intuitive meaning of the different temporal operators is depicted on Figure 4. The states coloured black are those which satisfy expression. Each computation tree satisfies the CTL expression (above the tree).

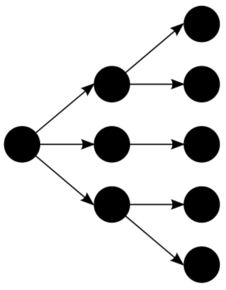
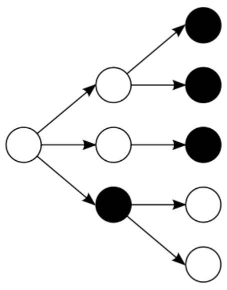
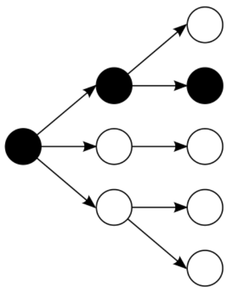
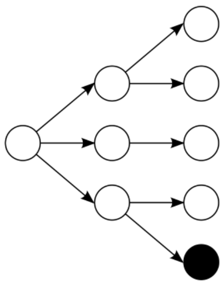
A[] «expression»	**A<> «expression»**	**E[] «expression»**	**E<> «expression»**
			

Figure 4. The intuitive meaning of the UPPAAL CTL operators

There are also two special operators:

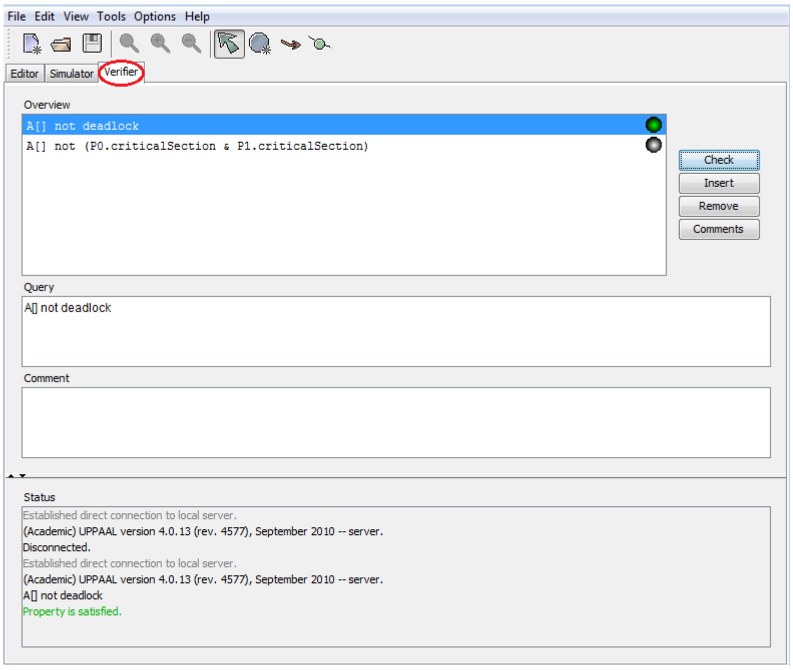
- A[] not deadlock: true, if there is no deadlock in the system.
- «logical expression1» --> « logical expression2»: true, if after satisfying logical expression1 the system goes eventually to a state (in every branch), where logical expression2 is satisfied.

For example, the following specification requirement is a valid expression: A[] d1.money>=1000. This property is true only if the system contains an instance *d1* where the value of variable *money* will never be less than 1000.

It is important to note that the expressions are not independent from each other, according the rule of duality:

- ¬(A[]«logical expression ») = E<> ¬« logical expression »
- ¬(A<>« logical expression ») = E[] ¬« logical expression »

We can check the CTL specification expressions in UPPAAL in the *Verifier* module (tab). It can be seen on Figure 5. We can define the temporal logic expressions in the *Query* textbox, which can be checked by choosing the *Check* button. If the model satisfies the requirement, the circle next to the expression (in the "*Overview*") becomes green; if the specification does not hold, this circle becomes red (and grey if the expression has not been evaluated yet).



```
A[] not deadlock
A[] not (P0.criticalSection & P1.criticalSection)
```

Query: A[] not deadlock

```
Established direct connection to local server.
(Academic) UPPAAL version 4.0.13 (rev. 4577), September 2010 -- server.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.0.13 (rev. 4577), September 2010 -- server.
A[] not deadlock
Property is satisfied.
```

Figure 5. UPPAAL Verifier

We can set at the menu at the tab of *Options* the *Diagnostic trace* from *None* to something else (see Figure 6.) (for example *Shortest*), then the tool produces an example or a counterexample for our specification, which can be seen in the *Simulator* view. It helps finding the problems with our model, useful for debugging.

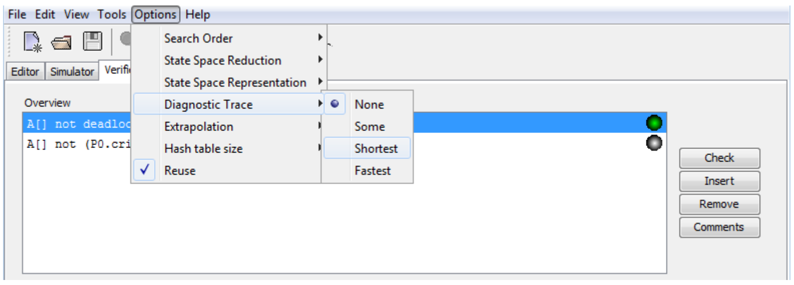


Figure 6. Setting the diagnostic trace

For the precise definition of the syntax of the expressions and for more examples we refer the reader to [3].

Now, you can express your specification in UPPAAL! You can check whether the (system) model contains the following problems: Race conditions, Starvation, Deadlock, Livelock. Design the specification in the CTL language of UPPAAL and verify it!

References

[1] Gerd Behrmann, Alexandre David, and Kim G. Larse: A Tutorial on Uppaal 4.0
http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf

[2] UPPAAL2k: Small Tutorial
http://www.it.uu.se/research/group/darts/uppaal/tutorial.pdf

[3] Gerd Behrmann: Introduction to UPPAAL
http://people.cs.aau.dk/~srba/courses/SV-05/slides/l11.pdf