SSY191 - Model-based development of cyberphysical systems
Individual Assignment 2

Lucas Rath

Problem 1

A) The C-code presented represents the Petri net behaviour accordingly. In the main function, three tasks are created, which will run in parallel and independently. Each task represents one of the three tokens that start in the state $p0$ and the execution of certain block of commands, which are omitted in the C-code, represents the correspondent token being in a certain state.

The additional tokens in r_A and r_B are represented as semaphores `resources_a` and `resources_b` respectively. In the same way as it is in a Petri-net, one transition depends on the availability of one token/semaphore to proceed. The `Take` command, represents a transition that needs one resource (r_A or r_B) and one token $p*$ (represented by a task). Similarly, `Give` is a transition that gives back the token to r_A or r_B.

However, there is one deadlock in the system, which happens when one Task is state p_1 and other at state p_4. This situation can be reproduced for the following sequence of events:

```
Task_1    (p0-->p1)   take (r_a)
Task_1    (p1-->p2)   take (r_b)
Task_1    (p2-->p3)   -
Task_1    (p3-->p4)   give (r_a)
Task_2    (p0-->p1)   take (r_a)
```

In this way, `Task_1` can not proceed because it requests `r_a` that was taken by `Task_2`. `Task_2` is also blocked because it needs `r_b`, which was taken by `Task_1`. Finally, `Task_3` can not proceed because it is requesting `r_a`, taken by `Task_2`.

B) The resource-allocation graph for the deadlock situation can be seem in Figure 1. Evidently, this is a deadlock situation because the graph contains a cycle and there is only one instance per resource type. Therefore there must be a deadlock.

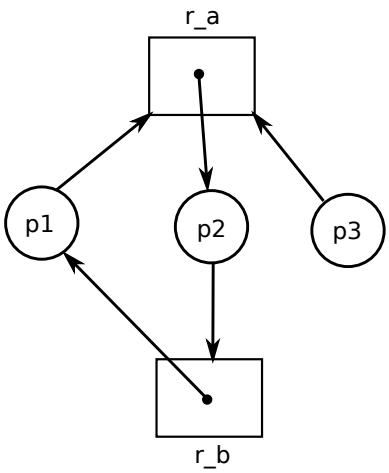

Figure 1: *Resource-Allocation Graph showing deadlock state.*

Clearly, to occur a deadlock, there must exist a cycle in the RAG. In a deadlock situation, one or more processes are blocked requesting a resource that is taken by another process, which in turn is also

blocked requesting another resource, taken by another process that is blocked, and so on. Clearly, this must constitute a cycle, because all blocked states are requesting a resource from other process which is also blocked requesting. This chain of blocked states are therefore always in a loop. The opposite is however not always the case, because not all cycle situations imply in a deadlock.

C) Once identified, the solution to avoid the deadlock is to increase additional restrictions to the system to avoid certain combination of states. In particular we want to avoid one processes being at state p1 and other at p4 at the same moment. We then add an additional resource `r_c`, which will avoid this situation. The solution is represented as a Petri-net in Figure 2 and the code is shown in Listing 1.

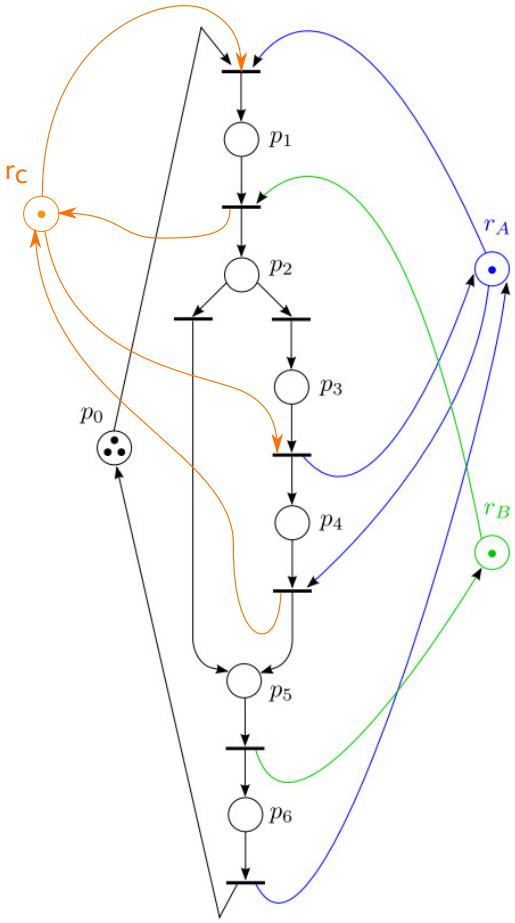

Figure 2: *Petri-net without deadlocks.*

Listing 1: *C-code using the FreeRTOS API for running tasks that have the same behavior is the Petri-net in Figure 2*

```
#include <stdio.h>
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"

xTaskHandle task_a_Handle;
xTaskHandle task_b_Handle;
xTaskHandle task_c_Handle;

SemaphoreHandle_t resource_a;
SemaphoreHandle_t resource_b;
SemaphoreHandle_t resource_c;

void the_task(void *pvParameters)
{
    while (1)
    {
        xSemaphoreTake(resource_c, portMAX_DELAY);  // Take r_c to enter p1
        xSemaphoreTake(resource_a, portMAX_DELAY);
        ...
        xSemaphoreTake(resource_b, portMAX_DELAY);
        xSemaphoreGive(resource_c, portMAX_DELAY);  // Give r_c to enter p2
        ...
        if (...)
        {
            ...
            xSemaphoreTake(resource_c, portMAX_DELAY); // Take r_c to enter p4
            xSemaphoreGive(resource_a);
            ...
            xSemaphoreTake(resource_a, portMAX_DELAY);
            xSemaphoreGive(resource_c, portMAX_DELAY); // Give r_c to enter p5
            ...
        }
        xSemaphoreGive(resource_b);
        ...
        xSemaphoreGive(resource_a);
        ...
    }
}

int main(int argc, char **argv)
{
    resource_a = xSemaphoreCreateMutex();
    resource_b = xSemaphoreCreateMutex();
    resource_c = xSemaphoreCreateMutex();
    xTaskCreate(the_task, "Task 1", configMINIMAL_STACK_SIZE, NULL, 1, &
        task_a_Handle);
    xTaskCreate(the_task, "Task 2", configMINIMAL_STACK_SIZE, NULL, 1, &
        task_b_handle);
    xTaskCreate(the_task, "Task 3", configMINIMAL_STACK_SIZE, NULL, 1, &
        task_c_handle);

    vTaskStartScheduler();
    for( ;; );
}
```