Uppgift 2

The Smoker’s Problem

Parallella processer G1F

Grupp A4

Adam Näslund a13adana  
Victor Karlsson c12vicka  
Alexander Milton b13alemi

# 1 Inledning

Grunduppgiften bestod i att simulera en situation där ett antal individer står samlade runt ett bord. En av individerna är en agent och de övriga är rökare. Varje rökare har en oändlig mängd av en av de tre resurser som krävs för att röka, vilka är tobak, papper och tändstickor. Agenten har en oändlig mängd av samtliga resurser och lägger upp en enhet var av två olika slumpmässigt valda resurser på bordet. När de kompletterande resurser den behöver finns på bordet tar en rökare resurserna från bordet och börja röka. När en rökare tagit resurserna från bordet lägger agenten fram nya resurser.

I grunduppgiften finns en rökare av varje sort, det vill säga en som har tobak, en som har papper och en som har tändstickor. Två utökade versioner av grundproblemet gavs även, i den första finns ett godtyckligt antal rökare och i den andra finns dessutom krav på att en lösning garanterar en rättvis fördelning av resurserna (fairness) samt att alla rökare får röka någon gång (liveness).

Samtliga problem i uppgiften löstes med hjälp av programmeringsspråket SR och lösningen på det första problemet modellerades och verifierades även i verktyget UPPAAL.

# 2 Problembeskrivning

*Smoker’s Problem* beskriver en situation där ett antal rökare och en agent samarbetar för att producera cigaretter som rökarna sedan röker. Agenten förfogar över en oändlig mängd av de tre resurser som krävs för att producera en cigarett vilka är tobak, papper och tändstickor. Rökarna har en oändlig mängd av en av de tre resurserna och behöver en enhet var av de andra två för att kunna röka.

Agenten placerar slumpmässigt en enhet var av två olika resurser på ett bord. En rökare som har den kompletterande resursen tar resurserna från bordet och röker. När bordet är tomt lägger agenten fram nya resurser på bordet.

## 2.1 Grundproblem

I grundproblemet finns tre rökare, en som har tobak, en som har papper och en som har tändstickor. Det finns ingen problematik som rör konkurrens om företräde mellan rökarna då resurstypen skapar mutual exclusion mellan dem. Det finns heller inga krav på rättvis fördelning (fairness) eller på att alla rökare får röka (liveness).

Global invariant för en lösning av detta problem försäkrar att agenten placerar ut två olika resurser.

* ∀i,j ∈ resources(onTable(i) ∧ onTable(j) → typeOf(i) ≠ typeOf(j))

## 2.2 Utökat problem: Godtyckligt antal rökare

I den första utökade versionen av problemet finns det ett godtyckligt antal rökare. Det kan därmed finnas flera rökare av varje sort och lösningen måste garantera att resurserna enbart delas ut till en rökare. Det kan även finnas ingen rökare av en viss typ, och programmet måste stödja möjligheten att en resurskombination aldrig efterfrågas utan att deadlock uppstår. Det finns dock fortfarande inga krav på fairness eller liveness i lösningen.

## 2.3 Andra utökade problem: Fairness och liveness

I den andra utökade versionen av problemet ställs, utöver tidigare krav, även krav på att alla rökare garanterat får röka (liveness genom eventual entry) och att lösningen måste vara rättvis (fairness). Rättvisa kan tolkas på olika sätt men den tolkning som valdes var att samtliga rökare får röka lika många gånger; det vill säga att om det finns r antal rökare och agenten lägger fram resurser n gånger får varje rökare röka minst ⌊r/n⌋ gånger.

# 3 Metodik

## 3.1 Design

Uppgiften är en variation på det klassiska problemet Producer/Consumer där två olika typer av processer existerar, en typ som producerar resurser och en som konsumerar de producerade resurserna. I det här fallet finns en producent i form av agenten och tre olika typer av konsumenter i form av rökare.

### 3.1.1 Grundproblem

Grundproblemet löstes genom att varje rökare har ett index som identifierar dem samt anger vilken typ av resurs som hålls av rökaren. Fyra kanaler skapas, en för varje typ av rökare och en för agenten. Rökarnas kanaler används av agenten för att meddela vilka resurser den lagt på bordet och agentens kanal används av rökarna för att skicka begäran om resurser när de mottagit signal om att deras respektive resurstyper placerats på bordet.

Agenten slumpar fram två olika resurser, beräknar vilken typ av rökare som saknar de resurserna och skickar en signal på motsvarande kanal. Rökaren som lyssnar på den kanalen tar emot signalen från agenten och skickar en signal för att begära tillgång till resurserna. När agenten tagit emot signalen om efterfrågan töms bordet, rökaren konsumerar resurserna och samtliga processer återgår till sitt utgångsläge och två nya resurser slumpas fram.

### 3.1.2 Utökat problem: Godtyckligt antal rökare

I den andra deluppgiften har den första lösningen utvecklats med ett antal fundamentala skillnader. För att ett arbiträrt antal rökare skall kunna röka måste programmet se till att agenten enbart producerar resurser som behövs. Agenten har fortfarande en egen kanal för att te emot begäran om resurser och varje rökare har sin egen, unika kanal för att ta emot bekräftelser.

Istället för att agenten omedelbart börjar producera resurser så skickar varje rökare ett meddelande på agentens kanal innehållandes dess ID samt vilken typ av resurs denne redan innehar. Agentens kanal buffras då med meddelanden likt en FIFO-kö och behandlas ett i taget. Agenten producerar de två resurser som efterfrågats av rökaren vars meddelande ligger först i kön och skickar sedan en signal till denne om att resurserna är redo.

Slutligen töms bordet, rökaren konsumerar sina resurser och agenten behandlar nästa efterfrågan i kön. Rökare som vill ha resurser och inte redan står i kön hamnar automatisk sist i kön när de skickar sin efterfrågan. Denna lösning kringgår till viss del uppgiftskraven i det avseende att resurserna inte längre slumpas fram, utan väljs ut specifikt beroende på rökarnas behov. I annat fall skulle en framslumpad kombination av resurser som inte efterfrågas av någon rökare orsaka dödläge, då de aldrig kan lämna den kritiska sektionen (bordet).

### 3.1.3 Andra utökade problem: Fairness och liveness

I den tredje och sista deluppgiften måste handhavandet av resurserna ske under definierat rättvisa förhållanden och lösningen garantera liveness. Uppgiftsbeskrivningen specificerar att agenten slumpmässigt ska välja vilka resurser som ska läggas på bordet. Detta är ej förenligt med liveness då det kan innebära att rökare av en särskild typ svälts eftersom de resurser de inte behöver placerats på bordet eller att inga rökare av en specifik typ existerar trots att motsvarande resurser lagts fram. Slumpmässigheten innebär även att fairness inte kan garanteras eftersom att vissa resurser kan komma att läggas upp flera gånger än andra. Det var därför nödvändigt att frångå den delen av uppgiftsbeskrivningen för att kunna garantera liveness och fairness i lösningen.

I den tredje deluppgiften har inga ändringar skett från föregående uppgiftslösning. I och med att resurserna väljs ut och distribueras helt beroende på vilka rökare som efterfrågar resurser med hjälp av en FIFO-kö är lösningen helt rättvis, förutsatt att rättvisa i situationen definieras enligt problembeskrivningen; om det finns r antal rökare och agenten lägger fram resurser n gånger får varje rökare röka minst ⌊r/n⌋ gånger.

Om exempelvis 100 rökare efterfrågar 100 kombinationer av resurser, samma eller olika, kommer agenten efter 100 iterationer ha betjänat alla 100 rökare en gång vardera, trots att alla rökare i praktiken skulle kunna begära exakt samma resurser. Även om den första rökaren omedelbart efterfrågar nya resurser efter att ha rökt kommer denne bli den 101:a rökaren att betjänas i kön. Eftersom att slumpelementet frångåtts kan fairness alltså uppnås.

Ett potentiellt hinder i lösningsdesignen till detta problem är att agenten saknar medvetenhet om hur många rökare som existerar, vilket kan ge upphov till race conditions (i form av först-till-kvarn-distribution) från busy wait eller förlorad fairness. Med vår lösning krävs enbart att agentens buffert i meddelandekanalen är tillräckligt stor för att tillåta för lika många buffrade meddelanden som antalet rökare som existerar. Så länge ett meddelande garanterat når sin mottagare garanterar programmet att rättvis distribution av resurser och liveness hos processerna genom eventual entry till den kritiska sektionen.

### 3.2 Verifiering

*In this section, you shall discuss which properties that your solution must have to be correct. ou shall also discuss how you are going to verify each of these properties, e.g., which tests to run and which queries to give the model checker.*

Den första lösningen hade som krav att det fanns exklusiv tillgång till resurserna på bordet mellan processerna. Efter att de placerats ut kan de endast plockas upp en gång utav en rökare (mutual exclusion).

## 4 Resultat

### 4.1 Implementation

*The source code need not be included in the report since you are supposed to submit the sr files to your drop box. However, this is the place where you can discuss the code you have written. Do you have any comments to your source code?*

För att leva upp till sina krav frångår andra och tredje lösningarna beskrivningen på två sätt. Agenten slumpar inte vilka resurser som läggs på bordet och agenten lägger inte upp resurserna oberoende av rökarna, detta skulle krävt komplex kod för att hantera situationer då ingen rökare av en särskild typ finns samt skulle varit omöjligt att förena med den tredje lösningens krav på fairness och liveness.

Även om den inte specifikt utformades för detta uppfyller den andra lösningen både liveness och fairness då alla rökare köar i en FIFO-kö i agentens meddelandebuffer. Därför fanns ingen anledning att skriva någon särskild kod för den tredje lösningen utan koden för den andra lösningen användes.

### 4.2 Verifiering

*This section contains your arguments for your solution meeting, or not meeting, the requirements. Are you confident with your solution? With what inputs did you test the program? To what extent have you used uppaal for verification? What queries did you run and what was the result of this? List your queries with the results you got from the model checker and explain what these results mean. For example: A[] !deadlock, result true, means that you have absence of deadlock. Given the verification, under what circumstances can you be sure that your program is correct?*

Den första lösningens korrekthet verifierades genom att den modellerades i UppAal och att följande frågor ställdes mot modellen med positivt resultat.

* A.Initial --> A.BroadcastInfo

Verifierar liveness i agenten, efter att den startats kommer den kontakta en rökare.

* E<> (S1.Smoke || S2.Smoke || S3.Smoke)

Verifierar att någon rökare kommer få röka.

* A[] !A.CalculateSmokerType || (A.first\_item != A.second\_item)

Verifierar att de två resurser som läggs på bordet alltid är av olika typ.

* A[] !((A.smoker\_type > 3) && (A.smoker\_type < 1))

Verifierar att agenten alltid kommer fram till en giltig typ på den rökare den tänker skicka till.

* A[] !deadlock

Verifierar att lösningen aldrig kan hamna i deadlock.

De krav som fanns den första lösningen var mutual exclusion vid tillgång till bordet samt att lösningen ej skulle kunna hamna i deadlock. Då endast agenten har tillgång till bordet och det endast finns en agent garanteras mutual exclusion vid tillgång till bordet. Att lösningen inte kan hamna i deadlock verifierades via en förfrågning gentemot UppAal-modellen.

Att modellen överensstämmer med koden för första lösningen är troligt då både koden och modellen är enkla och lättförståeliga. Därmed kan lösningen med tämligen stor sannolikhet sägas uppfylla de krav som ställts på den.

De andra två lösningarna verifierades inte via modeller utan enbart genom testning, men även för dem bidrar deras enkelhet och förståelighet till ökat förtroende deras korrekthet.

# 5 Diskussion

*This section contains a discussion of whether the problem was successfully solved. Have you identified any problems with your solution? Can your solution be improved and if so how? Was it anything that surprised you during the verification? Note: it’s seldom a good approach to conclude that the program is perfect. I want to see that you can estimate your success.*

Lösningen till första deluppgiften kan inte garantera liveness då den väljer resurser slumpmässigt vilket kan resultera i att processer svälts om de resurser de behöver ej slumpas fram. Lösningen hanterar även bara en rökare av varje sort; fler av samma typ leder till att flera rökare tilldelas samma resurser och om det inte saknas rökare av någon typ riskerar lösningen att hamna i deadlock. Den löser dock det angivna problemet, som inte tar hänsyn till dessa problem.

Även om den andra lösningen inte utformades med målet att ha liveness eller fairness resulterade designen i att den hade både dessa egenskaper. Därmed kunde den användas som lösning även på det tredje problemet. Till skillnad från den första lösningen väntar agenten på att rökarna begär resurser innan den skapar dem, vilket på ett enkelt och elegant sätt ger både fairness och liveness via agentens meddelandebuffer. Lösningen frångår dock problembeskrivningen något, då den anger att resurserna skall väljas ut slumpmässigt, något som omöjliggör rättvis distribuering av resurser och således även liveness och fairness.

De angivna queries (i den första deluppgiften) godkänns av UPPAAL Model Checker, vilket påvisar att vår lösning är fri från deadlock samt garanterar mutual excluson samt eventual entry hos agenten.