Universidade do Minho
Escola de Engenharia

Programação Ciber-Física 2023/2024

Trabalho Prático 2

Ricardo Lopes Santos Silva :: pg54188; Afonso Xavier Cardoso Marques :: pg53601


I. Introdução

Neste trabalho, abordaremos a modelação e análise de um sistema ciber-físico utilizando a linguagem de programação Haskell, com ênfase no uso de monads. Monads são uma poderosa abstração em Haskell que permitem tratar efeitos como estado, I/O, e não-determinismo de uma forma funcional e elegante. A tarefa proposta envolve a modelação de um cenário específico onde quatro aventureiros precisam de atravessar uma ponte de corda, respeitando restrições de segurança e tempo. Através desta modelação, pretendemos demonstrar a viabilidade de cumprir os requisitos temporais impostos, além de garantir que todas as regras de segurança sejam respeitadas.

O relatório elaborado explica o código desenvolvido e as conclusões obtidas durante a análise. Este exercício não só ilustra a aplicação prática das monads em Haskell, mas também destaca a importância da modelação precisa em sistemas ciber-físicos complexos.

A. *Contextualização*

No meio da noite, quatro aventureiros encontram uma ponte de corda velha que atravessa um desfiladeiro profundo. Por razões de segurança, decidem que não mais do que duas pessoas devem atravessar a ponte ao mesmo tempo e que uma lanterna precisa de ser carregada por um deles em cada travessia. Eles têm apenas uma lanterna. Os quatro aventureiros não são igualmente habilidosos: atravessar a ponte leva-lhes 1, 2, 5 e 10 minutos, respetivamente. Um par de aventureiros atravessa a ponte em um tempo igual ao do mais lento dos dois aventureiros.

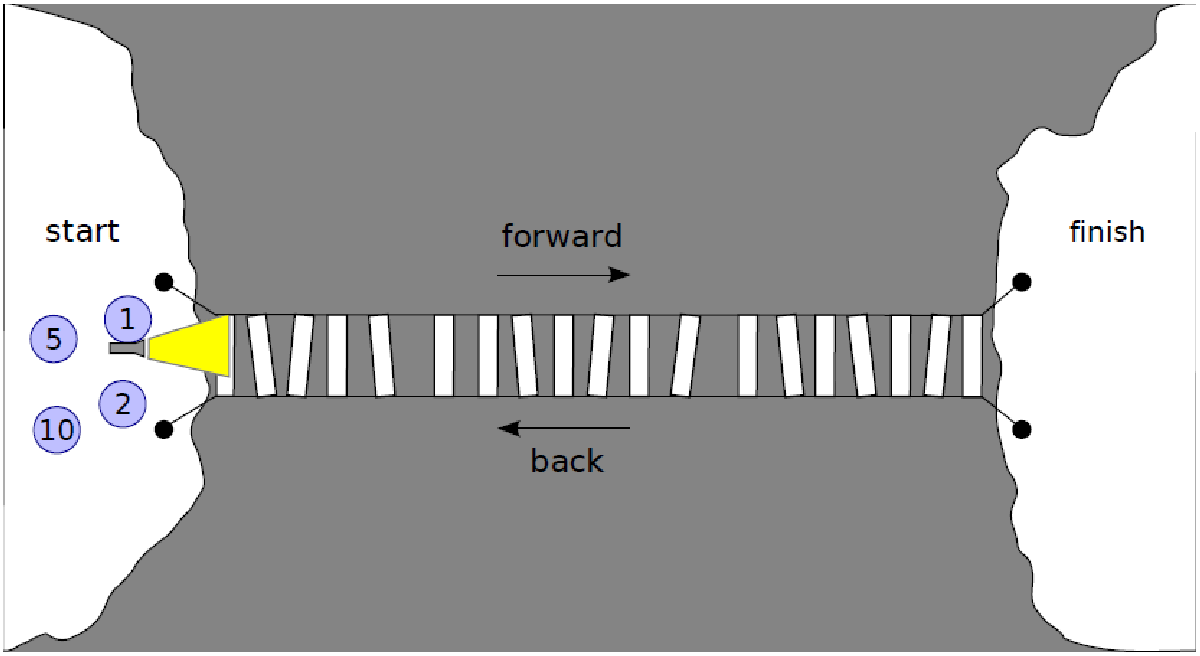

Fig. 1: Cenário em estudo

Um dos aventureiros afirma que eles não podem todos estar do outro lado em menos de 19 minutos. Um companheiro discorda e afirma que isso pode ser feito em 17 minutos.

II. Exercício 1

A primeira tarefa é verificar as afirmações seguintes utilizando Haskell. Especificamente, será preciso:

1) **modelar** o sistema acima usando **monads**, em particular as de duração e não-determinísticas;
2) mostrar que é realmente **possível** que todos os aventureiros estejam do outro lado em 17 minutos;
3) mostrar que é **impossível** que todos os aventureiros estejam do outro lado em menos de 17 minutos.

Para cumprir esta tarefa, completou-se o código no anexo (Adventurers.hs), ou seja, adicionou-se uma definição às funções que carecem de uma definição, seguindo os comentários presentes no código. Para auxilio à resolução, usou-se como inspiração o monad de duração dos slides e do código Haskell que foi previamente fornecido pelo docente. Também se analisou detalhadamente o código referente ao *Knight's quest* e, em particular, a monad LogList.

*1) **Código desenvolvido**:*

Analisando o código fornecido, notamos que o tipo Objects é declarado da seguinte forma:

```

type Objects = Either Adventurer ()
```

Tendo em conta o contexto do problema, onde um aventureiro pode trazer a lanterna consigo ou não, e a definição de Objects, que é um par composto por um aventureiro no lado esquerdo e um dado indefinido no lado direito, fica claro que esse dado indefinido deverá ser a lanterna. Sendo assim, adicionamos as seguintes declarações sobre a lanterna:

```

-- Utilities
-- lanterna
lamp :: Objects
lamp = Right ()

-- mover a lanterna
moveLamp :: State -> State
moveLamp = changeState lamp
```

Onde *lamp* é definido como sendo o lado direito de um par Objects, e a movimentação da lanterna envolve a mudança do seu estado. De forma a tornar a resolução mais adequada ao código fornecido, criaram-se as definições *getTimeOAdv* e *adventurers* onde os aventureiros são tratados como um par Objects.

```

-- igual a getTimeAdv só que para os objetos
getTimeOAdv :: Objects -> Int
getTimeOAdv (Left a) = getTimeAdv a

-- lista dos aventureiros como objetos
adventurers :: [Objects]
adventurers = map Left [P1, P2, P5, P10]
```

getTimeOAdv apenas devolve o resultado de *getTimeAdv* para o lado esquerdo de Objects e *adventurers* devolve uma lista de Objects, onde aplica a função Left a cada elemento da lista [P1, P2, P5, P10], isto é, cria uma lista de valores do tipo Either Adventurer (), onde cada elemento é do tipo Left Adventurer.

Quanto às funções fornecidas para completar, vamos abordar a estratégia implementada. Começamos então pela função **getTimeAdv**:

```

getTimeAdv :: Adventurer -> Int
getTimeAdv P1  = 1
getTimeAdv P2  = 2
getTimeAdv P5  = 5
getTimeAdv P10 = 10
```

Esta função é de implementação bastante simples. Serve apenas para indicar o tempo que cada aventureiro demora a passar a ponte. De seguida, também é requisitada a definição da função **allValidPlays**, que para um certo estado devolve todas as jogadas possíveis dos aventureiros.

```

allValidPlays :: State -> ListDur State
allValidPlays s = manyChoice [moveOne s, moveTwo s]
```

Tendo em conta que a função recebe um estado *s* como argumento, e quer-se saber quais são todas as jogadas possíveis para *s*, temos de ter em conta se apenas se vai mover um aventureiro ou dois ao mesmo tempo. Para isso foram criadas as funções auxiliares *moveOne* e *moveTwo*.

```

moveOne :: State -> ListDur State
moveOne s = LD (map f canCross)
  where
    -- canCross :: [Objects]
    canCross = filter ((== s lamp) . s) adventurers
    -- f :: State -> Duration State
    f x = wait (getTimeOAdv x) (return (changeState x (moveLamp s)))


moveTwo :: State -> ListDur State
moveTwo s = LD (map f po)
  where
    -- pares dos aventureiros que podem atravessar a ponte
    -- po :: [(Objects, Objects)]
    po = makePairs (filter ((== s lamp) . s) adventurers)

    -- duracões dos pares após atravessarem
    -- f :: (Objects, Objects) -> State
    f (x, y) = wait (max (getTimeOAdv x) (getTimeOAdv y)) (return (changeState y (changeState x (moveLamp s
    ))))
```

- *moveOne* recebe o estado como argumento e devolve todas os estados atingíveis a partir de *s* juntamente com a duração de cada um. Para esse efeito, precisa de filtrar todos aventureiros que podem atravessar a ponte, ou seja, todos os que estão no mesmo lado que a lanterna (têm o mesmo estado) e de seguida calcula o tempo da travessia para cada um (aplicando em conjunto as mudanças de estado).

- *moveTwo* faz exatamente a mesma coisa, com a particularidade de ter em conta que são dois aventureiros em vez de um, logo, após a filtragem dos aventureiros com estado igual à lanterna, criam-se todos os pares de aventureiros possíveis para a travessia, sendo que a duração da travessia vai ser o valor do aventureiro que demora mais tempo.

Como as duas funções, *moveOne* e *moveTwo*, devolvem uma lista de durações de estados (ListDur), é necessário juntar essas duas listas numa só, ficando apenas uma única lista com todas os movimentos possíveis. Recorremos para isso à função *manyChoice*, disponibilizada no código do docente, que constrói uma lista de durações a partir da concatenação de listas de durações de menor dimensão.

Além destas funções, também nos foi pedido para implementar as funções **exec**, **leq17** e **l17**. A função **exec**, dado um valor inteiro *n* e um estado inicial, calcula todas as *n* possíveis sequências de movimentos que os aventureiros podem fazer. Para isso, ...

```

exec :: Int -> State -> ListDur State
exec 0 s = return s
exec n s = do s' <- allValidPlays s
              exec (n-1) s'
```

Sendo que ListDur State é igual a [Duration State], ou seja, uma lista de durações para cada estado, a implementação desta função foi para dois casos diferentes:

1) Caso base:
   Quando o número de passos é 0, a função simplesmente retorna o estado atual *s*, encapsulado em um monad ListDur.

2) Caso recursivo:
   Para um número de passos *n* maior que 0, a função **allValidPlays s** gera todos os estados válidos que podem ser alcançados a partir do estado atual *s*. O resultado é um monad que encapsula uma lista de novos estados com as respetivas durações. Em seguida, usando a notação *do*, extraí-se cada novo estado *s'* da lista gerada por allValidPlays s. Após isso, a função **exec (n-1) s'** é chamada recursivamente. O processo mantem-se até que *n* se torne 0. A recursividade permite explorar todos os possíveis caminhos dentro do número de passos especificado.

A função **leq17** verifica a possibilidade de todos os aventureiros passarem para o lado direito em 17 ou menos segundos, em 5 jogadas.

A lógica para a função **le17** é semelhante no entanto verifica apenas a possibilidade de todos passarem em menos de 17 segundos num intervalo de jogadas definido por nós.

*2) **Testes e Resultados Obtidos:*** Quanto à parte da testagem, tivemos que desenvolver algoritmos...

III. EXERCÍCIO 2

A segunda tarefa consiste em comparar ambas as abordagens (via UPPAAL e Haskell) para o problema dos aventureiros. Especificamente, o objetivo é fornecer os pontos fortes e fracos das duas abordagens: quais são as (des)vantagens do UPPAAL para este problema? E quanto ao Haskell?

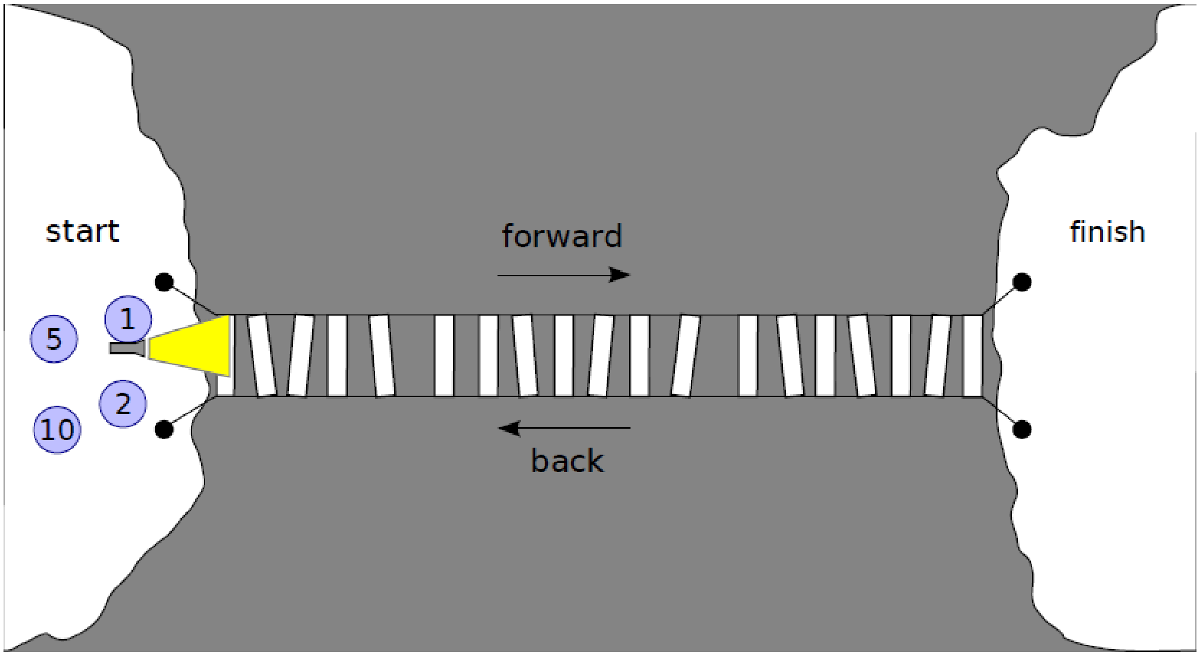

Fig. 2: Cenário em estudo no UPPAAL

De um modo geral, cada abordagem tem as suas vantagens e desvantagens, como é de esperar.

Na abordagem via Uppaal, conseguimos verificar e validar o modelo, bem como obter uma análise minuciosa dos seus potenciais pontos de falha. Isso ocorre porque ao executar simulações, o Uppaal analisa aleatoriamente todos os casos possíveis e identifica possíveis *deadlocks* no programa. As simulações também são um ponto forte do Uppaal em relação ao Haskell, pois sua execução é visual e auxilia no melhor entendimento do verdadeiro comportamento modelado, passo a passo. Isso resulta em uma correção melhor e mais rápida de quaisquer comportamentos inesperados. No Uppaal também não implica diretamente algoritmos de programação, pois é possível simular problemas complexos utilizando as ferramentas simples fornecidas pelo Uppaal e aproveitar essas ferramentas utilizando formas lógicas para verificar quaisquer requisitos do sistema. Podemos verificar facilmente as propriedades temporais de um modelo a partir de fórmulas CTL.

IV. Conclusão

Com este projeto terminado, apresentamos uma breve conclusão que consideramos englobar todo o processo de aprendizagem despoletado por este trabalho. Em suma, consideramos que o trabalho desenvolvido e os resultados apurados são positivos, demonstrando uma clara capacidade em aplicar os conceitos das aulas de Programação Ciber-Física. Destacamos como pontos positivos as soluções encontradas para cada um dos cenários e em particular o cumprimento das regras de cada um. No entanto, reconhecemos que existem alguns pontos a melhorar, tais como aumentar a complexidade de alguns dos autómatos, permitindo assim que as propriedades que não se conseguiram provar nas duas fases exploradas passassem a ser satisfazíveis. Como trabalho futuro, seria também interessante introduzir cenários ainda mais complexos, que envolvessem, por exemplo, mais do que duas estradas, introduzindo assim um novo conjunto de desafios.

Appendix A
Simulação do UPPAAL na parte 1

images/parte_1/P1_simulation.png

APPENDIX B
RESULTADOS VERIFIER DO UPPAAL NA PARTE 1

images/parte_1/P1_verifier.png

APPENDIX C
SIMULAÇÃO DO UPPAAL NA PARTE 2

images/parte_2/P2_simulation.png

APPENDIX D
RESULTADOS VERIFIER DO UPPAAL NA PARTE 2

images/parte_2/P2_verifier.png

REFERENCES

[1] https://uppaal.org/features/
[2] https://haslab.github.io/MFP/PCF/2324/index
[3] https://www.youtube.com/watch?v=7yDmGnA8Hw0
[4] https://www.haskell.org/
[5] https://en.wikipedia.org/wiki/Monad_(functional_programming)