Universidade do Minho

Departamento de Informática

Programação Ciber-Física

Modelação e análise de um sistema ciber-físico com mónadas

Trabalho Prático 2

Carlos Ferreira PG47087
Henrique Neto PG47238

20 de junho de 2022

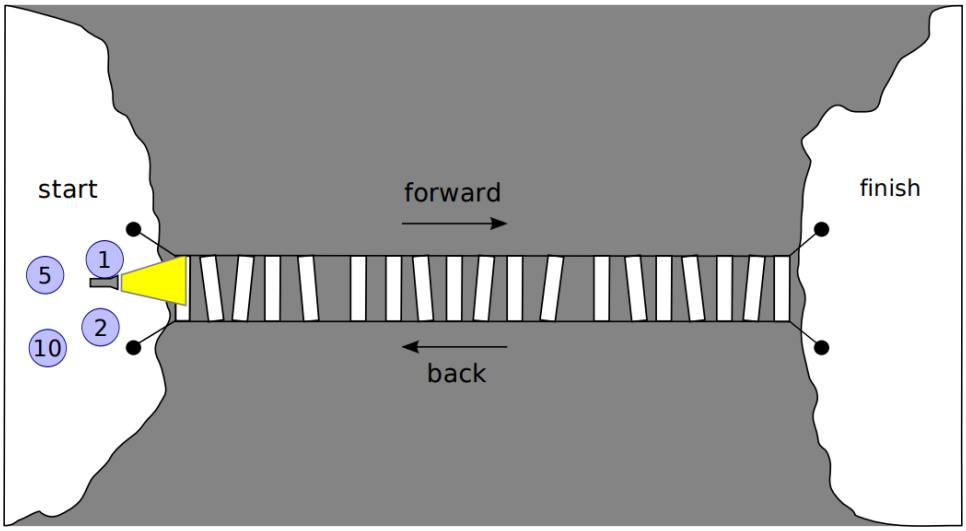
1 Introdução

O objetivo deste segundo trabalho prático é homologo ao objetivo do primeiro, sendo que ambos visam modelar e analisar sistemas que simulem problemas ou situações restringidos por um conjunto de regras comportamentais e envolventes de fatores físicos, nomeadamente o tempo.

Ambos os projetos diferenciam nas metodologias usadas, pelo que neste segundo trabalho será usada a linguagem *Haskell* que fornece características e capacidades que muito diferenciam da ferramenta anteriormente utilizada *UPPAAL*.

Tem-se então como finalidade simular e responder a um problema básico denominado de quatro aventureiros que consiste na seguinte situação: Quatro aventureiros encontram-se frente a uma ponte muito desgastada, pelo que no máximo dois aventureiros a podem atravessar ao mesmo tempo, tendo sempre na sua presença uma lanterna durante a passagem. Existe somente uma lanterna e como problema adicionar cada aventureiro contém diferentes níveis de habilidade pelo cada um demora um diferente intervalo de tempo a atravessar a ponte, no caso de 2 elementos atravessarem a ponte simultaneamente o intervalo de tempo gasto será igual ao tempo gasto pelo aventureiro com pior habilidade, o objetivo obviamente é que todos os aventureiros se encontrem no outro lado da ponte.

Após a modelação do sistema se encontrar completa pretende-se que este consiga responder a questões, como por exemplo, a possibilidade de todos os aventureiros atravessarem a ponte em menos de um determinado tempo.



Este relatório consiste em três diferentes partes, numa primeira parte pretende-se fornecer e explicar o código em *Haskell* que consiga responder ao problema expondo o conceito de Mónadas e as vantagens desta que tornam a modelação do sistema eficaz. Numa segunda parte irá-se fazer uma comparação entre as diferentes metodologias usadas em ambos os projetos da disciplina, tentando-se demonstrar as vantagens e desvantagens observáveis na utilização das ferramentas *UPPAAL* e *Haskell*. Por fim, a terceira fase tem como intuito expandir a Mónada implementada, para assim conseguir aumentar as capacidade de resposta e analise do problema em questão, demonstrando a versatilidade do uso da técnica.

2 Primeira Parte

2.1 Monads

Nesta secção explicamos a estrutura e finalidade das mónadas usadas nesta modulação.

2.1.1 Monad Duration

Esta mónada implementa a descrição de um fator temporal às execuções. Para tal este agrega a cada valor um intervalo de tempo. Com isto é possível a partir desta construção associar a cada execução a sua respetiva duração, que pode ser definida e/ou incrementada a partir do método **wait**. Por fim, as instâncias monádicas permitam que consigamos de forma arbitrária compor várias durações, sendo que o resultado da composição monádica possui a duração obtida pela soma de todas as durações e o valor obtido pela composição dos valores.

```
module DurationMonad where

-- Defining a monad (the duration monad) --

data Duration a = Duration (Int, a) deriving Show

getDuration :: Duration a -> Int
getDuration (Duration (d,x)) = d

getValue :: Duration a -> a
getValue (Duration (d,x)) = x

instance Functor Duration where
  fmap f (Duration (i,x)) = Duration (i,f x)

instance Applicative Duration where
  pure x = (Duration (0,x))
  (Duration (i,f)) <*> (Duration (j, x)) = (Duration (i+j, f x))

instance Monad Duration where
    (Duration (i,x)) >>= k = Duration (i + (getDuration (k x)), getValue (k x))
    return x = (Duration (0,x))

wait :: Int -> Duration a -> Duration a
wait i (Duration (d,x)) = Duration (i + d, x)
```

2.1.2 Monad ListDur

Esta mónada representa uma coleção de durações. Com isto a função `return` devolve uma coleção correspondente a um *singleton* da duração do elemento enquanto que a composição monádica compõem cada duração da coleção fornecida com cada elemento obtido da **ListDur** resultante de aplicar a função obtida ao valor em questão.

Adicionalmente, sendo isto uma coleção de durações, existe a necessidade de possuir várias funções auxiliares de forma a tirar partido da informação contida. Desta forma, desenvolvemos 5 funções para esta finalidade.

isEmpty indica se a coleção está vazia;

filterDuration filtra os elementos da coleção com base num predicado dado;

anyDuration indica se existe um elemento da coleção que satisfaz o predicado dado;

allDuration indica se todos os elementos da coleção satisfazem o predicado dado;

composeDurations compõem cada elemento da coleção com a função dada;

```
data ListDur a = LD [Duration a] deriving Show

remLD :: ListDur a -> [Duration a]
remLD (LD x) = x

isEmpty :: ListDur a -> Bool
isEmpty = null . remLD

filterDuration :: (Duration  a -> Bool) -> ListDur a-> ListDur a
filterDuration f = LD . filter  f . remLD

anyDuration :: (Duration  a -> Bool)-> ListDur a -> Bool
anyDuration f = any  f . remLD

allDuration :: (Duration  a -> Bool)-> ListDur a -> Bool
allDuration f = all  f . remLD

composeDurations :: (a -> Duration  b) -> ListDur a -> ListDur b
composeDurations f = LD . map (>>= f) . remLD

instance Functor ListDur where
   fmap f l = LD (map (fmap f) (remLD l))

instance Applicative ListDur where
   pure x = LD [pure x]
   l1 <*> l2 =  LD [f <*> x | f <- remLD l1, x <- remLD l2 ]

instance Monad ListDur where
   return = pure
   l >>= k =  LD $ do x <- remLD l
                      map (\d -> x >>= const d ) (remLD (k (getValue x)))

manyChoice :: [ListDur a] -> ListDur a
manyChoice = LD . concatMap remLD
```

2.2 Modelação do problema

Esta secção é relativa ao código desenvolvido para simular o comportamento do problema em questão, tendo como fundamento as definições das Mónadas demonstradas na secção anterior.

2.2.1 Funções básicas

Para a modelagem do sistema foi necessário então representar três entidades diferentes assim como as suas respetivas funções básicas que permitem retirar informação destas entidades ou altera-las conforme as transações do problema.

Adventurer

A entidade aventureiro representa obviamente os elementos que necessitam atravessar a ponte, pela qual, como dito no enunciado, existem quatro aventureiros distintos P1,P2,P5 e P10, cada um com diferente nível de habilidade. É importante observar que foi vantajoso usando a linguagem *Haskell* derivar esta estrutura em **Ord** por razões que serão discutidas posteriormente.

allAdventurers Esta declaração consiste na lista dos diferentes 4 aventureiros.

getTimeAdv Esta função simula os distintos níveis habilidade pelo que faz a correlação entre cada aventureiro e o respetivo tempo que este demora a atravessar a ponte.

Objects

A entidade objeto expande os elementos que atravessam a ponte, pela qual, assim como mencionado no enunciado é adicionado aos aventureiros a existência de uma lanterna, sendo necessário avaliar também a localização desta. Para a adição da lanterna foi utilizada a construção **Either**, fazendo a distinção entre objetos que são do tipo aventureiros, representados por **Left** ou a lanterna representada por **Right**.

lantern Esta declaração representa o objeto lanterna para efeitos de melhor compreensão por parte de quem interpreta o código desenvolvido.

allAdventurersObjects Esta declaração representa a lista de todas entidade aventureiro mas em formato objeto, sendo estes projeções de todas as entidades **Adventurer** efetuadas com a função **Left**.

allObjects Declaração da lista de todos os objetos do sistema.

getTimeObject Função que expande **getTimeAdv**, ou seja, dando um objeto, devolve o tempo que este demora a percorrer a ponte. Visto que durante as travessias a lanterna é sempre acompanhada por um aventureiro, e o tempo de travessia é igual ao elemento com pior habilidade, por convenção diz-se que o tempo de travessia da lanterna é nulo.

State

A entidade **State** que representa o sistema na sua totalidade, faz a correlação entre cada objeto e a sua localização, sendo o lado esquerdo da ponte representado pelo booleano **False**, e o lado direito representado por **True**.

Show and Eq Desta entidade, foi necessário derivar uma instância **Show** para que esta possa ser demonstrada pelo compilador. Tendo em conta a implementação desta, o código irá imprimir uma lista de booleanos, sendo o primeiro correspondente à posição da lanterna e os restantes à posição dos aventureiros. Adicionalmente, foi necessário também derivar **Eq** para que se possa calcular a igualdade entre dois estados e assim auxiliar varias operações futuras como por exemplo, calcular se todos os aventureiros se encontram no final da ponte.

gInit and gEnd Estas duas declarações representam respetivamente a situação inicial e final do problema, assim dizendo, todos os elementos se encontram no lado esquerdo ou direito da ponte.

changeState and mChangeState A função **changeState** tem como objetivo representar a transição de um único objeto para o lado oposto em que se encontra, devolvendo o estado resolvente de realizar essa transição. A função **mchangeState** realiza o mesmo conceito mas para a travessia de vários objetivos.

```
-- The list of adventurers
data Adventurer = P1 | P2 | P5 | P10 deriving (Show, Eq, Ord)

-- List of all of the possible adventures
allAdventurers :: [Adventurer]
allAdventurers = [P1, P2, P5, P10]

-- The time that each adventurer needs to cross the bridge
getTimeAdv :: Adventurer -> Int
getTimeAdv P1 = 1
getTimeAdv P2 = 2
```

```haskell
getTimeAdv P5 = 5
getTimeAdv P10 = 10

-- Adventurers + the lantern
type Objects = Either Adventurer ()

-- The lantern
lantern :: Objects
lantern = Right ()

-- List of all of the possible adventur objects
allAdventurersObjects :: [Objects]
allAdventurersObjects = map Left allAdventurers

-- List of all of the possible objects
allObjects :: [Objects]
allObjects = lantern:allAdventurersObjects

-- The time that a object needs to cross the bridge
getTimeObject :: Objects -> Int
getTimeObject = either getTimeAdv (const 0)

instance Show State where
  show s = (show . fmap (show . s)) allObjects

instance Eq State where
  (==) s1 s2 = all (\x -> s1 x == s2 x) allObjects

-- The initial state of the game
gInit :: State
gInit = const False

-- The final state of the game
gEnd :: State
gEnd = const True

-- Changes the state of the game for a given object
changeState :: Objects -> State -> State
changeState a s = let v = s a in (\x -> if x == a then not v else s x)

-- Changes the state of the game of a list of objects
mChangeState :: [Objects] -> State -> State
mChangeState os s = foldr changeState s os
```

2.2.2 Calculo de transações

Com a implementação da mónada e dos comportamentos básicos de cada entidade, o objetivo passa agora a simular as transações possíveis entre estados. A finalidade é então mapear todos os estados possíveis alcançar a partir de um estado inicial para cada n número de travessias realizadas. Observando que esta secção do relatório envolve termos mais técnicos e complexos do que a secção anterior, em vez de ser explicada cada função será apresentada inicialmente um conjunto de operações sequencias realizadas para que, a partir de um certo estado, consiga-se devolver os estados alcançáveis numa única travessia, indicando em que parte do código abaixo demonstrado se encontra cada operação.

1. Filtrar elementos

   Dado o estado ao qual se aplica a travessia, a primeira etapa consiste em recolher todos os aventureiros capazes de realizar a travessia, isto pois esta só é possível com a presença da lanterna. A realização desta operação encontra-se na função **validPlays** que filtra os aventureiros que se encontram no lado da lanterna.

2. Combinar aventureiros

   Após obter os aventureiros capazes de realizar a travessia, é sabido que no máximo 2 assim serão o capaz de fazer, pelo que, a partir da lista de aventureiros inicial é retirado todas as jogadas possiveis, ou seja uma lista de todas as combinações de 1 ou 2 aventureiros, como se pode ver em **adventurerPlays**. Estas listas representam na verdade conjuntos de aventureiros, sendo assim irrelevante a ordem em que é apresentada. Desta forma, para garantir que a construção do *Haskell* não nos devolve conjuntos iguais com ordens diferentes a entidade **Adventurer** foi ordenada com recurso a uma instância automática da classe **Ord**, de forma a que possamos exigir que as jogadas (listas) construias respeitem essa ordem. Com este pressuposto deixa de haver necessidade de pós-processamento para remover jogadas repetidas, sendo assim a implementação mais simples e eficiente. De seguida é necessário acrescentar a cada combinação a presença da lanterna sabendo que esta também realizará a travessia como se pode ver em **validPlays**.

3. Aplicar Transação

   Como se pode observar em **play**, para cada combinação de travessias de um possível estado é então aplicado a função previamente mencionada **mChangeState** para alterar as posições dos objetos devolvendo o estado resultante.

4. Aplicar passagem de tempo

   Depois desse novo estado ser convertido numa monad do tipo ListDur (return em **play**) é então necessário aplicar o aumento de tempo associado a respetiva transição realizada. Esta operação esta presente em **applyPlay** e o tempo a ser adicionado é calculado, como expresso no enunciado, a partir do máximo de tempo associado a cada aventureiro da combinação.

5. Agregar estados

   Para cada combinação calculada nas etapas 1 e 2 (**validPlays**), devem ser aplicadas as etapas 3 e 4 (**play**), pelo que no final os possíveis estados atingíveis a partir do estado inicial aplicando uma única transição devem ser agregados numa única monad **ListDur State**. Assim como a função **allValidPlays** explicitamente se comporta.

```
-- Lists all the adventures moves from the given list of objects
adventurerPlays  :: [Objects] -> [[Objects]]
adventurerPlays l = [[x] | x <- l, isLeft x] ++ [[x,y] |
x <- l, isLeft x, y <- l, isLeft y, x > y]
                        where isLeft = either (const True) (const False)

-- Lists all valid plays from the given state
validPlays :: State -> [[Objects]]
validPlays s = map (lantern:)
        (adventurerPlays (filter (\x -> s x == s lantern) allAdventurersObjects))

-- Applies the plays changes to the list of durations
applyPlay :: [Objects] -> ListDur a -> ListDur a
applyPlay os = let t = maximum (map getTimeObject os)
```

```
                    in composeDurations (wait t . return)

-- Moves the given Objects
play :: State  -> [Objects] -> ListDur State
play s p = (applyPlay p . return .  mChangeState p) s

{-- For a given state of the game, the function presents all the possible moves that the
adventurers can make.  --}
allValidPlays :: State -> ListDur State
allValidPlays s = manyChoice (map (play s) (validPlays s))
```

No entanto, como referido anteriormente, as etapas descritas acima somente nos permite ter em atenção a realização de uma única transição, pelo que, para poder calcular os estados alcançáveis efetuando um numero arbitrário de transições, é necessário realizar mais uma função, onde é possível observar uma clara vantagem na utilização de monad para modelagem de sistemas, sendo que as operações de agregação da estrutura que não são necessariamente relevantes no ponto de vista do problema são todas realizadas pela composição monádica implícita no *do*. A função **exec** devolve então uma estrutura **ListDur State** onde nela contém os estados alcançáveis a partir de **n** travessias aplicadas um estado inicial **s**, assim como as suas respetivas durações.

```
{-- For a given number n and initial state, the function calculates
all possible n-sequences of moves that the adventures can make --}
exec :: Int -> State -> ListDur State
exec n s= if n <= 0
          then return s
          else do newSs <- allValidPlays s
                  exec (n - 1) newSs
```

2.3 Verificações

Após a modelação do problema, prossegui-se com a validação deste, nomeadamente se este respeitava as propriedades descritas no enunciado como:

1. Mostrar que é possível todos os aventureiros se encontrarem no lado oposto da ponte em menos ou em 17 minutos.
2. Mostrar que é impossível todos os aventureiros se encontrarem no lado oposto da ponte em menos de 17 minutos.

Como se pode desde já prever, mostrar uma possibilidade normalmente é mais fácil pois só é necessário um caso onde tal se verifique, mostrar uma impossibilidade é muito mais complicado pois é necessário verificar que algo não é valido em todos os casos.

2.3.1 Alternativa 1

```
endedLeq :: Int -> ListDur State -> Bool
endedLeq i = anyDuration (\x -> (getDuration x <= i) && (getValue x == gEnd))

{-- Is it possible for all adventurers to be on the other side in <=17 min
and not exceeding 5 moves ? --}
leq17 :: Bool
leq17 =  any (endedLeq 17 . \x -> exec x gInit) [1..5]
```

Uma primeira alternativa passa por tirar partido da função **exec** e assumir um intervalo de travessias ao qual se pode procurar um estado no qual todos os aventureiros se encontram

no lado direito da ponte e a duração é menor que um certo valor. Se tal estado for encontrado, então é prova-se uma possibilidade.

As duas funções acima fazem exatamente isso: A função **endedLeq** procura numa lista de estados de duração, um que satisfaça as condições ditas em cima, a função **leq17** procura para um número de travessias entre 1 e 5, uma em que **endedLeq** é satisfeita para uma duração menor que 17. Realizando o comando **leq17** provou-se então que é possível todos os aventureiros se encontrarem no lado oposto da ponte em 17 minutos ou menos e também no máximo de 5 travessias.

Para provar uma impossibilidade com esta estratégia era necessário atribuir um número de iterações ao qual sabemos que se torna impossível satisfazer a condição, sabendo que a situação com mais iterações para o menor intervalo de tempo é quando o aventureiro mais rápido realiza a travessia consecutivamente. Para provar a impossibilidade para menos que 17 minutos era necessário testar aproximadamente 17 iterações, o que tendo em conta a implementação atual, demoraria demasiado tempo graças ao crescimento exponencial de iterações que o *exec* apresenta.

2.3.2 Alternativa 2

Surgindo então outra alternativa muito mais eficiente que a primeira para combater os problemas visíveis e assim ser capaz de testar eficazmente uma impossibilidade,

```
{-- Is it possible for all adventurers to be on the other side in < 17 min ? --}
l17 :: Bool
l17 =  execd (gEnd ==) (\d -> 17 > getDuration d) (return gInit)

{-- Searches for a execution from the given List of states that satisfies the first
and second predicate.Additionaly all states preceding the possible execution will
also satisfy the second predicate. --}
execd :: (State -> Bool) -> (Duration State -> Bool) -> ListDur State -> Bool
execd f g ls = let exec = filterDuration g (ls >>= allValidPlays)
        in not(isEmpty ls) && (anyDuration (f . getValue) exec || execd f g exec)
```

Esta alternativa passa por implementar uma função semelhante a **exec**, mas que na realização de cada iteração, realize um conjunto de operações de forma a restringir o seu comportamento e diminuir a carga computacional do metodo original.

O que realça esta alternativa da anterior é que esta descrimina os estados em que prosseguirá com a sua avaliação, iterando apenas sobre os estados em que existe a possibilidade de surgir um estado pretendido. Se esta avaliar todos os estados que possuem esta possibilidade e mesmo assim não encontre o estado pretendido, estamos perante uma impossibilidade.

Recorrendo ao exemplo dos 17 minutos, o que se realiza nesta alternativa é que para um dado **ListDur State** será aplicado a composição monádica com a função **allValidPlays** para estender a **ListDur State** inicial a todos os estados de duração da iteração seguinte. Após tal ser realizado, será feito um filtro a todos os estados que não tenham duração menor que 17, visto que esses casos já não precisam ser avaliados, pois das suas iterações já não serão encontrados estados com durações menores que satisfaçam o predicado desejado. Se dessa filtração resultar uma lista vazia estamos perante uma **impossibilidade** devolvendo **Falso**, se for encontrado num desses estados a condição que se procura estamos então perante uma possibilidade devolvendo **True**, se não ocorrer nenhum dos casos será feito a próxima iteração pelo uso da recursiva. Pela execução do comando l17 prova-se que é impossível que todos os aventureiros se encontrem no lado oposto da ponte em menos de 17 minutos.

3 Segunda Parte

3.1 *UPPAAL* vs *Haskell*

Durante a realização de ambos os projetos detetou-se bastantes diferenças entre as características que ambas as metodologias apresentam, sendo estas *Haskell* e UPAALL.

Neste capitulo do relatório será então apresentado as particularidades de ambas as abordagens em estudo, realçando para cada uma delas as vantagens e desvantagens encontradas na realização de problemas de modelação e análise de sistemas ciber-fisicos, sendo estas apresentadas de forma geral e em particular usando o problema dos quatro aventureiros.

Experiência

Na modelação de sistemas, em termos de velocidade de aprendizagem pode-se afirmar que o *UPPAAL* é uma ferramenta intuitiva de aprender por ser bastante legível e iterativa, tanto que mesmo sem qualquer experiência de utilização a modelação do problema foi trivial.

Em contra-partida o *Haskell* exige muito mais tempo para se dominar, podendo a modelação do problema ser complexa mesmo em casos onde já há conhecimento prévio como foi neste projeto.

Foco

UPAALL é uma ferramenta que se foca maioritariamente na implementação de sistemas distribuídos restritos por fatores temporais, dando importância a consequente comunicação entre as diferentes entidades do sistema, tirando partido do facto de ser totalmente possível garantir várias propriedades que verifiquem o bom funcionamento do nosso sistema distributivo usando invariantes.

A linguagem *Haskell* não apresenta foco em particular, pelo qual, vários tipos de problemas, sendo eles restringido por quaisquer fatores físicos como tempo, probabilidade, termodinâmica, sendo assim totalmente modeláveis pela ferramenta, isto garante uma versatilidade enorme, conseguindo expandir para propriedades não detetadas no *UPPAAL*.

No entanto, a implementações destas não é imediata e exige experiência e conhecimento, porém é bastante simplificada pela utilização de mónadas cujas vantagem permitem programar problemas independentemente do contexto, existindo uma alta reusabilidade e simplicidade do modelo. *Haskell* em si é relativamente bom quando se deseja encontrar uma solução para um problema via poder computacional.

Considerando que o problema dos quatro aventureiros não consiste num sistema distribuído ao qual se deseja garantir certas propriedades mas sim na procura de uma resolução, deduz-se que o *Haskell* apresenta mais vantagens nesta determinada situação.

Implementação

Haskell e *UPPAAL* apresentam métodos de implementação totalmente diferentes, pelo que, a segunda ferramenta exige a construção de um modelo visual onde se deve apresentar todos os estados e todas as transições possíveis entre cada estado, sendo que, para casos como o problema dos quatro aventureiros não ser possível tirar partido do facto de que cada transição é sistematicamente calculável a partir do estado anterior, enquanto que em *Haskell* só é necessário implementar a função do calculo de transições possíveis. Obviamente para o problema em particular o *Haskell* apresenta uma vantagem bastante superior.

Escalabilidade

Por razões mencionadas no item anterior, a presença de mais entidades e mais transações no nosso sistema, afetam de forma diferente ambas as abordagens.

Para o problema em questão, existem vários estados, ou seja, distribuição de aventureiros entre os dois lados da ponte, que o nosso sistema pode presenciar a cada dada iteração, e ainda mais transições possíveis entre cada estado, sendo que, sem uma estratégia em particular a sua implementação no *UPPAAL* pode-se tornar bastante mais problemática.

Analise e Verificação

Durante a analise dos modelos construidos por ambas as ferramentas, o *UPPAAL* apresenta uma linguagem CTL (*Computation Tree Logic*) muito intuitiva, capaz de detetar facilmente possíveis situações ou garantir invariantes no sistema, sendo também possível a visualização iterativa do funcionamento do seu funcionamento, garantindo assim uma grande capacidade de *debbug*, no entanto, seu poder expressivo é limitado como foi concluído no projeto anterior.

O *Haskell*, em contra partida não tem linguagem especifica tendo assim que implementar cada propriedade, sendo esta metodologia menos intuitiva, no entanto, em contra-partida possui um poder expressivo que permite analisar propriedades que a outra ferramenta não consegue.

No problema dos quatro aventureiros, as propriedades desejadas detetar, possível $<= 17$ e impossível < 17, são ambas realizáveis em cada ferramenta, sendo arguidamente mais fácil no UPAALL.

Rapidez de implementação

Para sistemas distribuídos que envolvem poucas transações e estados, a implementação no *UPPAAL* é quase instantânea, sendo possível imediatamente retirar conclusões sobre o sistema.

No entanto, para sistemas que não seguem esse modelo pode ser mais vantajoso o uso de *Haskell* principalmente na presença de experiência com a metodologia, ou reutilização de código existente, sendo que as monads em particular são bastante modulares e podem ser facilmente aplicáveis a diferentes sistemas sem qualquer alteração necessária.

Interpretação

Na interpretação do sistema, novamente, para problemas com poucas transições o UPAALL é muito mais vantajoso pois trata-se de uma componente visual, para problemas com transições sistemáticas o *Haskell* é mais favorecido podendo-se facilmente retirar as regras que governam o problema.

A tabela em baixo representa uma simplificação mais visual das diferenças encontradas para cada metodologia.

	UPPAAL	*Haskell*
Facilidade de aprendizagem	Melhor	Pior
Fatores Físicos do problema	Temporal	Implementável
Escalabilidade	Problemática	Sistemática
Velocidade de implementação.	Rápida	Depende da Experiência
Debbug e deteção de erros	Trivial	Implementável
Foco em segurança e correção	Própria	Implementável
Sistemas Distribuídos	Própria	Implementável
Transações entre Estados	Explicitas	Sistemáticas
Interpretação do modelo	Visual	Dedutível
Poder expressivo de analise	Restrito e Fácil	Implementável

4 Extra

De forma a permitir a visualização das jogadas, a monada de **ListDur** foi modificada de forma a associar a cada duração (e consequentemente a cada valor) uma string de *Log*. Desta forma a lista de durações passou a ser uma lista de pares *String-Duration*. Tendo isto em conta as funções que envolviam predicados de durações (**filterDuration**, **anyDuration** e **allDuration**) foram modificadas de forma a fazer o seu teste ao segundo valor de cada elemento da lista.

Adicionalmente no caso onde havia transformações/consultas também foram necessárias alterações, sendo que em transformações unárias (**composeDurations**, o funtor) a string era inalterada (sendo simplesmente acrescentado "$id\times$") enquanto que em transformação binárias (**composeDurations**, <*>, >>=), ou seja, em que agrupávamos duas *ListDur* as strings eram concatenas (estando envolvido o acréscimo de uma construção do tipo "$(++)\times$").

Relativamente à mónada em si, a função de **return** ainda instância um *singleton*, porém este é agora um par contendo a string nula e a duração obtida pelo return do elemento dado. Adicionalmente, como já foi dito anteriormente, a composição monádica (>>=) para além de compor as durações concatena o *log* de a cada elemento recebido da lista produzida com o *log* de cada elemento.

Foi adicionalmente criada uma função **logLD** que permite concatenar a cada log de cada elemento da lista uma string.

4.1 Monads

4.1.1 Monad ListDur

```
data ListDur a = LD [(String, Duration a)] deriving Show

remLD :: ListDur a -> [(String ,Duration a)]
remLD (LD x) = x

isEmpty :: ListDur a -> Bool
isEmpty = null . remLD

filterDuration :: (Duration  a -> Bool) -> ListDur a-> ListDur a
filterDuration f = LD . filter  (f . snd) . remLD

anyDuration :: (Duration  a -> Bool)-> ListDur a -> Bool
anyDuration f = any (f . snd) . remLD

allDuration :: (Duration  a -> Bool)-> ListDur a -> Bool
allDuration f = all (f . snd) . remLD

composeDurations :: (a -> Duration  b) -> ListDur a -> ListDur b
composeDurations f = LD . map (id >< (>>= f)) . remLD

logLD:: String -> ListDur a -> ListDur a
logLD s = LD . map ((++s) >< id) . remLD

instance Functor ListDur where
   fmap f l = LD (map (id >< fmap f) (remLD l))

instance Applicative ListDur where
   pure x = LD [([],pure x)]
   l1 <*> l2 =  LD [(s2 ++ s1 ,f <*> x) | (s1, f) <- remLD l1, (s2,x) <- remLD l2 ]
```

```
instance Monad ListDur where
   return = pure
   l >>= k =  LD $ do sx <- remLD l
                      g sx
                      where g (s,x) = map ((s++) >< (\d -> x >>= const d))
                                          (remLD (k (getValue x)))

manyChoice :: [ListDur a] -> ListDur a
manyChoice = LD . concatMap remLD

-- Minor Helper from Cp
(><) :: (a -> b) -> (c -> d) -> (a,c) -> (b,d)
(><) f g (a, c) = (f a, g c)
```

4.2 Calculo de transações

O implementação anterior encontra-se completamente igual, podendo esta ser usado com esta mónada nova sem qualquer diferença. Porém esse caso não iria preencher informações nos logs, ficando este assim vazios. Logo para podermos tirar partido da nova estrutura, foi criado um método que permite codificar uma jogada numa *String* (**playFormat**). Adicionalmente o método **applyPlay** foi expandido, sendo que após este complementar as mesmas tarefas que já fazia antes irá acrescentar aos **logs** da coleção a codificação da jogada realizada ao chamar **logLD** na estrutura resultante.

```
-- Format's a play
playFormat:: [Objects] -> String
playFormat = ('>':) . concatMap (either ((' ':) . show) (const ""))

-- Applies the plays changes to the list of durations
applyPlay :: [Objects] -> ListDur a -> ListDur a
applyPlay os = let t = maximum (map getTimeObject os)
                   s = playFormat os
                   in logLD s . composeDurations (wait t . return)
```

5 Conclusão

A realização deste projeto permitiu perceber as intrínsecas complexidades envolventes na modulação e posterior análise de sistemas comprometidos por determinadas restrições físicas, que levam a regras e comportamentos que são necessários implementar e representar nas ferramentas assim escolhidas para modelar estes sistemas.

Chegou-se novamente a anterior conclusão, que sistemas básicos com comportamentos fixos e reguláveis já demonstram dificuldades inesperáveis e cuidados a ter muito aquém dos expectáveis a primeira vista, realçando a dificuldade de modelação de sistemas muito mais complexos comparados ao desenvolvido neste projeto, assim como são exigidos na área de software.

Apesar da equipa ter experiência com o uso da linguagem *Haskell*, a oportunidade de abordar o problema de modelação de sistemas ciber-físicos usando esta metodologia demonstrou uma adaptabilidade e potencial surpreendente, cujas características de muito se diferenciam com a metodologia *UPPAAL* usada no projeto anterior.