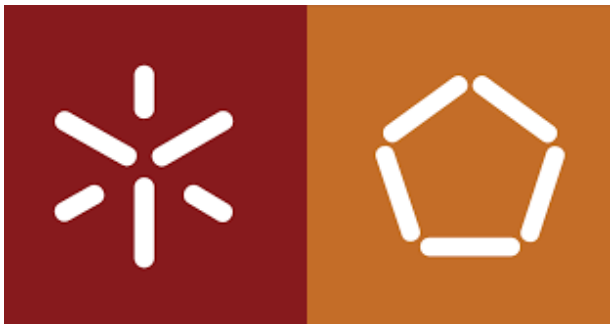

Universidade do Minho

Mestrado em Engenharia Informática
Programação Cíber-Física

Modelação e análise de um sistema ciber-físico
via UPPAAL e Haskell

Ana Henriques
pg50196

Teresa Fortes
pg50770

26 de junho de 2023

1 ª Tarefa

1.1 Modelação do sistema

A primeira tarefa proposta para este trabalho prático envolveu a **implementação de um modelo UPPAAL**, que simulasse um pequeno aeródromo privado. Esta parte procura explorar a aplicação de técnicas de modelação e verificação formal para a análise de sistemas reativos, com especial atenção aos detalhes exigidos pela partilha de recursos.

Numa fase prematura, considerámos apenas 2 aviões, que podem estar num destes quatro estados possíveis: **a voar** (`Flying`), **estacionado** (`Parked`), **a aterrar** (`Landing`) e **a levantar voo** (`TakingOff`). O campo para aterrar é um recurso partilhado pelos aviões, constituindo um ponto onde é necessário planear e controlar as atividades aéreas. Entretanto, o grupo estendeu o modelo e realizou as adaptações precisas para que ele fosse capaz de suportar um valor variável de aviões, representado por **N**.

Componente do avião

Conforme ilustrado na Figura 1, o estado inicial do avião é a voar. Contudo, este pode efetuar **um pedido para pousar**, o que aciona o canal `request`, uma ação complementar entre os modelos *Airplane* e *Controller*. Como forma de identificação, todas as ações do sistema recebem como argumento o ID do avião, um número inteiro que varia de 0 a N-1.

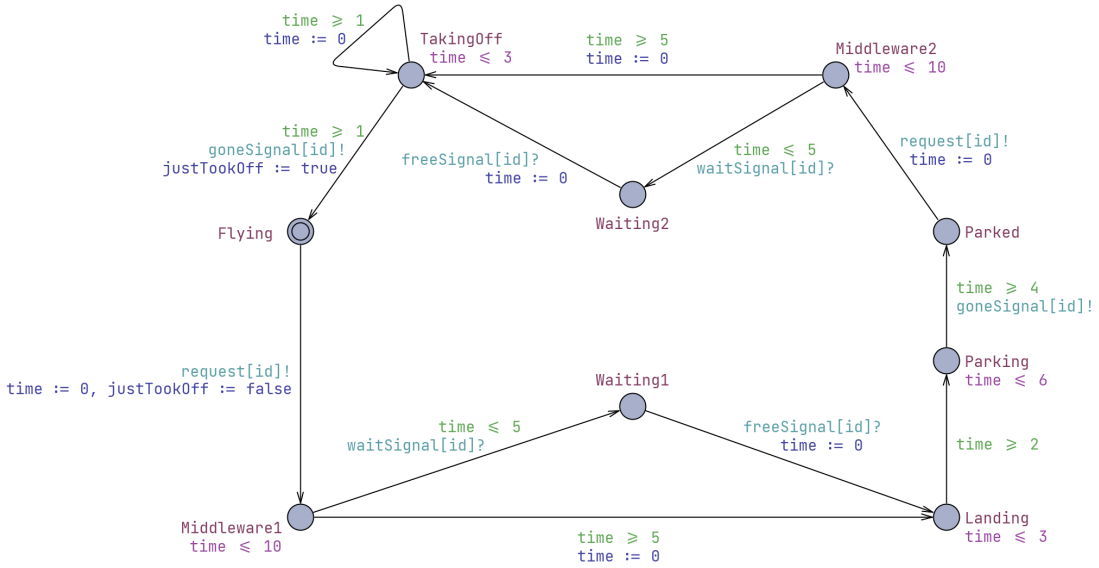

Figura 1: Modelo UPPAAL *Airplane*

Após 5 unidades de tempo sem qualquer aviso por parte do controlador para aguardar, o avião permanece no mesmo estado (`Middleware1`) por mais 5 unidades de tempo. No entanto, na situação do recurso compartilhado estar disponível, o avião prossegue com a execução da atividade de aterragem, que, em conjunto com a atividade de estacionamento, tem a duração de 4 a 6 unidades de tempo. Antes de chegar ao estado `Parked`, que representa o momento em que o veículo já está devidamente estacionado, o avião **envia o sinal de *gone*** ao controlador (canal `goneSignal`), para o informar acerca da conclusão da atividade e da liberação do recurso. Em contrapartida, se o recurso não estiver disponível,

o veículo é **solicitado a esperar** (canal `waitSignal`), mantendo-se em `Waiting1` até receber a notificação de que o campo ficou livre (canal `freeSignal`).

Assim que estacionado, o avião pode fazer um novo **pedido para levantar voo**, desencadeando a mesma sequência de execução adotada para a solicitação de aterragem. A única diferença reside no facto de que a atividade de descolagem tem a duração de 1 a 3 unidades de tempo. Quando esta atividade é concluída, o avião notifica o controlador com um sinal de *gone*, da mesma forma que ocorre na hora de pousar.

Componente do controlador

O componente destinado ao *Controller* do sistema, apresentado na Figura 2, tem início no estado `Idle`. Logo que receba uma solicitação de um avião (identificado por `e`, um ID), se não existir nenhum outro avião em corrente processamento (guarda: `len == 0`), o veículo é adicionado à lista de aviões que requisitaram a realização de uma atividade (*update*: `enqueue(e)`). A criação desta lista, de tamanho `len+1`, sendo `len` um inteiro que varia de 0 a N, foi uma das modificações estabelecidas para permitir que o sistema suporte N aviões. Esta adaptação foi feita com base no conceito de uma *queue*, o que possibilita controlar as entradas e saídas de pedidos e evitar conflitos na reserva do campo compartilhado pelos aviões. O primeiro elemento da lista é considerado **prioritário**.

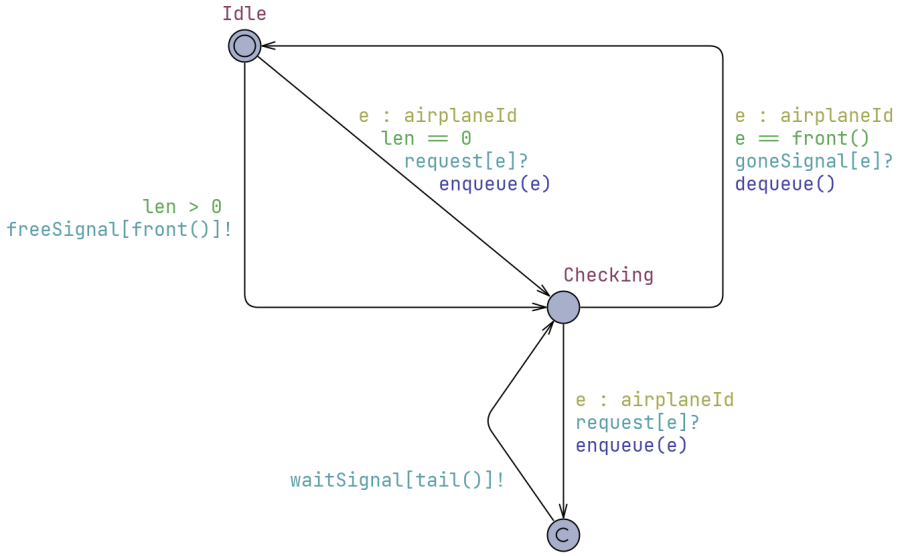

Figura 2: Modelo UPPAAL *Controller*

Ao avançar para o estado `Checking`, o controlador pode continuar a receber pedidos; porém, como os mesmos são colocados no fim da lista mencionada, o controlador enviar-lhes-á um sinal de espera até que o recurso volte a ficar disponível (canal `waitSignal[tail()]`). Retomando ao estado `Checking`, na eventualidade do avião em processamento terminar a sua atividade (canal `goneSignal[e]`, sendo `e` o primeiro pedido), o controlador remove esse elemento da lista (*update*: `dequeue()`) e retoma ao estado inicial. Caso haja aviões a aguardar tratamento (guarda: `len > 0`), o controlador autoriza o próximo veículo da lista a concretizar o seu pedido (canal `freeSignal[front()]`).

1.2 Verificação do modelo

Outro requisito proposto foi a definição de um conjunto de propriedades que o modelo deve satisfazer. Estas propriedades têm como objetivo garantir a segurança e eficiência das operações decorrentes no aeródromo, bem como a correta gestão do recurso compartilhado. Isto facilita na identificação de possíveis violações ou inconsistências presentes no sistema. O próximo passo é, então, a **verificação formal** das propriedades posteriormente enunciadas, recorrendo novamente à ferramenta UPPAAL.

Propriedades de alcançabilidade

- Um avião pode alcançar a aterragem.

```
E<> (exists (i:airplaneId) Airplane(i).Landing)
```

- Um avião pode alcançar a descolagem.

```
E<> (exists (i:airplaneId) Airplane(i).TakingOff)
```

- O Controller pode receber e guardar pedidos de aviões.

```
E<> Controller.Checking
```

- Um avião pode estar a aterrar enquanto todos os outros estão à espera.

```
E<> Airplane(0).Landing && (forall (i:airplaneId) i != 0 imply
      (Airplane(i).Waiting1 || Airplane(i).Waiting2))
```

Propriedades de segurança

- Não pode haver mais do que um avião a aterrar ao mesmo tempo.

```
A[] forall (i:airplaneId) forall (j:airplaneId) Airplane(i).Landing
      && Airplane(j).Landing imply i == j
```

- Se um avião esperar 5 unidades de tempo e não tiver recebido um sinal de espera, então deve permanecer no mesmo estado por mais 5 unidades de tempo.

```
A[] (Airplane(0).time > 5 && Airplane(0).Middleware1) imply
      (Airplane(0).Middleware1 && Airplane(0).time <= 10)
```

- Um avião deve demorar, no mínimo, 4 unidades de tempo para aterrar e estacionar.

```
A[] forall (i:airplaneId) Airplane(i).Parked imply
      Airplane(i).time >= 4
```

- Um avião deve demorar, no máximo, 6 unidades de tempo para aterrar e estacionar.

```
A[] forall (i:airplaneId) (Airplane(i).Landing or Airplane(i).Parking)
      imply Airplane(i).time <= 6
```

- Um avião deve demorar, no mínimo, 1 unidade de tempo para levantar voo.

```
A[] forall (i:airplaneId) (Airplane(i).Flying && justTookOff)
      imply Airplane(i).time >= 1
```

- Um avião deve demorar, no máximo, 3 unidades de tempo para levantar voo.

```
A[] forall (i:airplaneId) Airplane(i).TakingOff imply
      Airplane(i).time <= 3
```

- Não pode haver o sobrecarregamento da lista com N aviões.

```
A[] Controller.list[N] == 0
```

- O sistema nunca entra num estado de *deadlock*.

```
A[] not deadlock
```

Propriedades de *liveness*

- Se já existir um pedido de avião em corrente processamento, todos os pedidos que eventualmente chegarem ficarão à espera.

```
(Controller.Checking && Controller.len > 1) -> (forall (i:airplaneId)
      i != Controller.list[0] imply
           (Airplane(i).Waiting1 || Airplane(i).Waiting2))
```

- Sempre que um avião à espera de aterrar for o próximo da lista, eventualmente ele poderá aterrar.

```
(Airplane(0).Waiting1 && Controller.list[0] == 0) -> Airplane(0).Landing
```

- Sempre que um avião à espera de descolar for o próximo da lista, eventualmente ele poderá levantar voo.

```
(Airplane(0).Waiting2 && Controller.list[0] == 0) -> Airplane(0).TakingOff
```

- Sempre que o recurso compartilhado estiver disponível quando um avião pedir para aterrar, eventualmente ele poderá aterrar.

```
(Airplane(0).Middleware1 && Controller.len == 1) -> Airplane(0).Landing
```

- Sempre que o recurso compartilhado estiver disponível quando um avião pedir para descolar, eventualmente ele poderá descolar.

```
(Airplane(0).Middleware2 && Controller.len == 1) -> Airplane(0).TakingOff
```

- Sempre que um avião estiver a aterrar, eventualmente ele será estacionado.

```
Airplane(0).Landing -> Airplane(0).Parked
```

2 ª Tarefa

2.1 Diferenças entre modelação, verificação e programação

A modelação e a verificação, em contraste com a programação, desempenham papéis distintos no processo de desenvolvimento de sistemas, cada um com os seus próprios objetivos. Estas abordagens complementam-se e são fundamentais na criação de soluções eficazes e confiáveis.

A **modelação** envolve a criação de modelos abstratos que representam requisitos e interações do sistema, oferecendo uma visão conceitual do problema. Já a **verificação** passa por analisar e validar os modelos gerados na etapa anterior. Recorrendo, por exemplo, à lógica e à matemática, esta fase verifica se os modelos atendem aos requisitos estabelecidos, identificando erros ou inconsistências antes da implementação prática.

Em contrapartida, a **programação** é responsável pela implementação prática dos modelos previamente construídos. Nesta fase, o código fonte é escrito numa determinada linguagem de programação, seguindo as regras e sintaxe adequadas. A programação traduz, assim, as ideias subjacentes aos modelos em instruções executáveis pelo computador.

A modelação e a verificação ocorrem, por norma, antes da programação, tendo uma posição determinante na garantia da qualidade do resultado final. Enquanto que a modelação proporciona uma compreensão geral do contexto do problema, a verificação economiza tempo e recursos ao auxiliar na deteção de possíveis erros antes da construção do código. Por sua vez, a programação transforma as ideias num produto funcional. No entanto, em diversos cenários, os processos de modelação e de programação são executados em conjunto, de forma iterativa, a fim de assegurar a constante evolução do sistema. Isto porque, ao mesmo tempo que a modelação cede uma definição clara e precisa das funcionalidades à outra etapa, também a programação permite o aprimoramento dos modelos, à medida que ajusta a implementação de acordo com as necessidades reais do sistema.

2.2 Aplicação de exemplos concretos

O artigo feito por Xintao et al [1] propõe a verificação de modelos em tempo real, utilizando o *model checker* `UPPAAL-SMC` para avaliar a sua validade quando aplicados na área médica de traqueotomia a laser[1]. O caso de estudo tem, então, como objetivo garantir que **(1)** o laser seja apenas ativado quando a operação for segura e que **(2)** a concentração de oxigénio no paciente não esteja muito alta, de maneira a evitar queimaduras nos tecidos. Além disso, é imprescindível garantir que **(3)** a sua ventilação não caia para níveis perigosos durante o corte, e que **(4)** o laser emita luz por tempo suficiente para não interromper desnecessariamente o procedimento cirúrgico.

2.2.1 Modelação do sistema

O modelo é composto por 4 componentes diferentes: ***Patient***, referente ao paciente submetido a cirurgia; ***Ventilator***, referente ao ventilador que regula a respiração do paciente; ***Laser Scalpel***, referente ao laser que efetua o corte na traqueia; e ***Supervisor***, referente ao responsável por assegurar os requisitos de segurança da cirurgia.

[1] Uma cirurgia realizada em pacientes com problemas respiratórios.

Nas Figuras 3, 4, 5 e 6, podemos ver que o **ventilador** regula a respiração do paciente. Enquanto isso, os sinais fisiológicos do **paciente** (estados `Inhale`, `Exhale` e `Hold`) são medidos por sensores e enviados para o supervisor, que os analisa (canal `Start`). Se o **supervisor** aprovar o uso do **laser** (canal `SupervisorAppr`), o ventilador é interrompido. Caso contrário, o ventilador continua a funcionar normalmente (canais `VentHold`, `VentPumpIn` e `VentPumpOut`). Quando o corte aprovado é concluído (canal `SupervisorStop`), o ventilador volta a operar por instrução do supervisor. O laser pode ser, no entanto, ativado por meio de um pedido feito pelo cirurgião ao supervisor (canal `SurgeonReq`). Uma vez aceite, o laser é acionado e o cirurgião pode desligá-lo (canal `SurgeonStop`) ou cancelar a aprovação (canal `SurgeonCancel`) se for necessário tomar outras medidas.

Componente do paciente

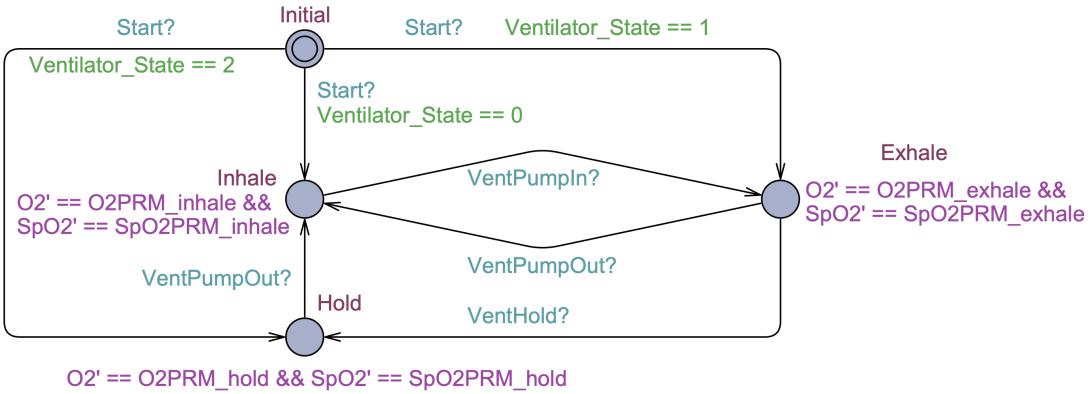

Figura 3: Modelo UPPAAL *Patient*

Componente do ventilador

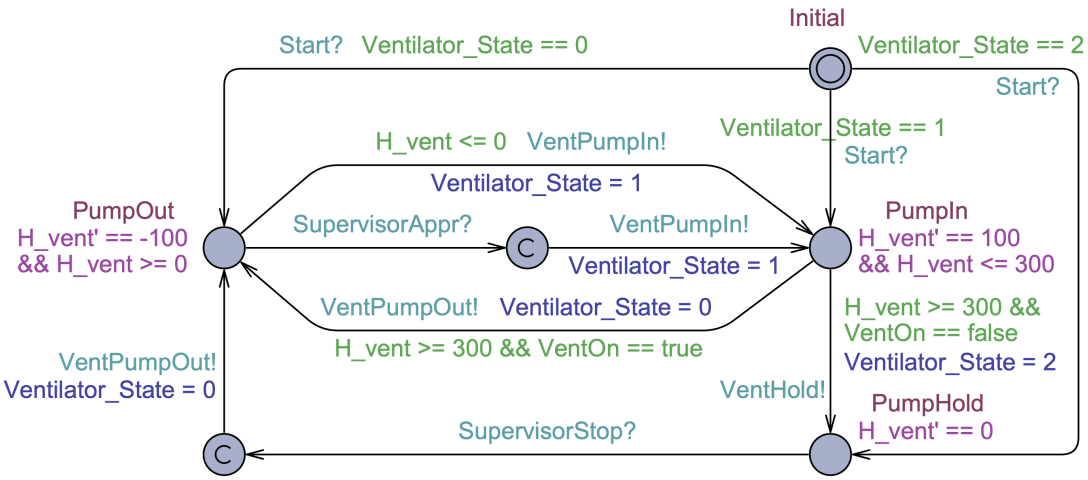

Figura 4: Modelo UPPAAL *Ventilator*

Componente do laser

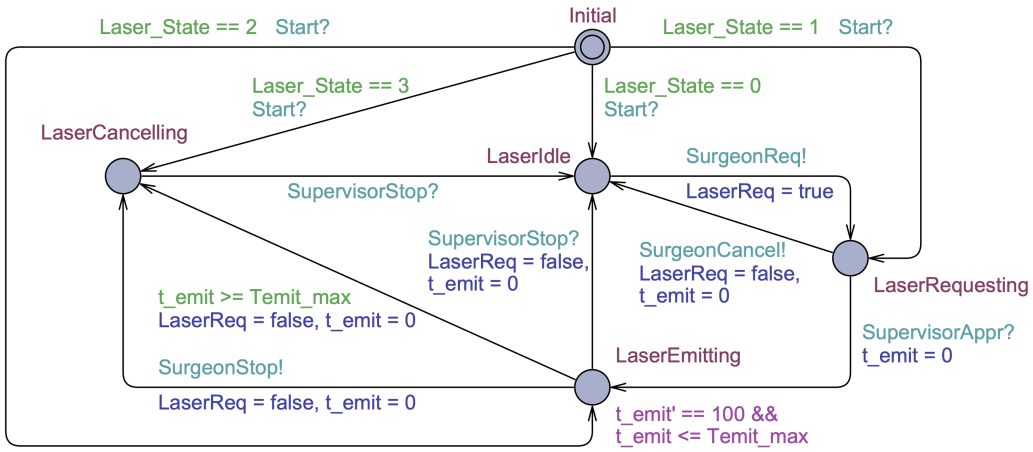

Figura 5: Modelo UPPAAL *Laser Scalpel*

Componente do supervisor

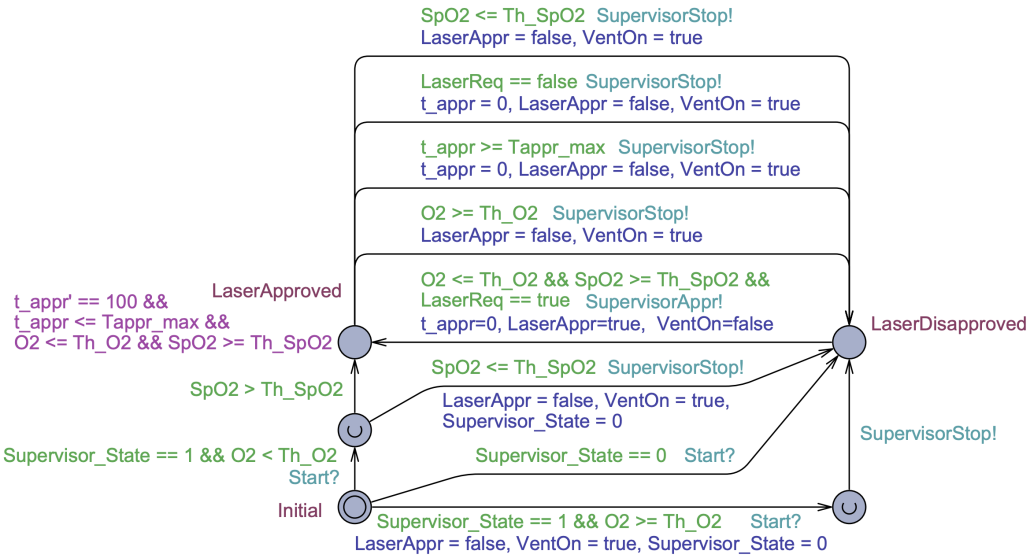

Figura 6: Modelo UPPAAL *Supervisor*

2.2.2 Verificação do modelo

Considerando estes requisitos, o artigo sugere, como propriedades de verificação, as apresentadas a seguir em $\texttt{WMTL}_{\leq}$[2]. Estas propriedades caracterizam situações que **não devem ser alcançáveis** e, como tal, a probabilidade de acontecerem devem ser nulas.

- **(2)** Dentro de 100 unidades de tempo, verificar se, em algum momento, a concentração de oxigénio está acima do limite definido enquanto o laser emite luz.

```
Pr[<=100] (<> O2 > Th_O2 && LaserScalpel.LaserEmitting)
```

[2] *Weighted Metric Temporal Logic*

- **(3)** Dentro de 100 unidades de tempo, verificar se, em algum momento, a saturação de oxigênio no sangue está abaixo do limite definido enquanto o laser emite luz.

```
Pr[<=100](<> SpO2 < Th_SpO2 && LaserScalpel.LaserEmitting)
```

- **(4)** Dentro de 100 unidades de tempo, verificar se, em algum momento, o laser para de emitir luz precocemente, quando o nível de oxigénio está acima do limite ou o nível de oxigénio no sangue está abaixo do limite.

```
Pr[<=100](<> (O2 > Th_O2 || SpO2 < Th_SpO2) && t_appr < Th_appr &&
                        LaserAppr == true)
```

2.2.3 Implementação do sistema

O artigo estudado nesta secção não fornece qualquer implementação prática para o sistema proposto. Em contrapartida, o grupo desenvolveu um pequeno excerto de código em *Python*, para ilustrar como o processo de programação se pode tornar bastante simples ao seguir um esquema abstrato, que representa todos os requisitos e interações do sistema.

O código seguinte não apresenta exemplos de simulação. No entanto, ao criar classes que correspondem a cada um dos componentes do sistema e definir as variáveis utilizadas para verificar a segurança, podemos constatar que a modelação do problema facilita consideravelmente a sua tradução em instruções compreensíveis pelo computador.

Concluímos, por isso, que a modelação e a verificação permitem uma compreensão mais organizada e clara do **sistema como um todo**. Com o estabelecimento de uma estrutura lógica e a definição das relações entre os componentes, é possível visualizar e verificar o funcionamento do sistema antes mesmo de escrever o código concreto.

```
class Patient:
    def __init__(self):
        self.O2 = 0
        self.SpO2 = 0

class Ventilator:
    def __init__(self):
        self.is_pumping = False
        self.H_vent = 0

    def pump_in(self):
        self.is_pumping = True

    def pump_out(self):
        self.is_pumping = False

    def pump_hold(self):
        self.is_pumping = False

class LaserScalpel:
    def __init__(self):
```

```
            self.is_requested = False
            self.is_approved = False
            self.is_emitting = False

        def surgeon_request(self):
            self.is_requested = True

        def approve(self):
            self.is_approved = True

        def emit(self):
            self.is_emitting = True

        def stop_emit(self):
            self.is_emitting = False

        def surgeon_cancel(self):
            self.is_approved = False

    class Supervisor:
        def __init__(self):
            self.is_approval_revoked = False

        def check_safety(self, patient):
            if patient.O2 > Th_O2 or patient.SpO2 < Th_SpO2:
                self.is_approval_revoked = True

        def approve_laser(self, laser_scalpel):
            laser_scalpel.approve()

        def revoke_approval(self, laser_scalpel):
            laser_scalpel.cancel_approval()

        def stop_laser(self, laser_scalpel):
            laser_scalpel.stop_emit()

    Th_O2 = 80
    Th_SpO2 = 90

    patient = Patient()
    ventilator = Ventilator()
    laser_scalpel = LaserScalpel()
    supervisor = Supervisor()
```

3 ª Tarefa

3.1 Linguagem Probabilística

A terceira e última tarefa deste trabalho prático propõe explorar o desenvolvimento da nossa própria linguagem probabilística. De acordo com os conceitos discutidos nas aulas, esta tarefa foi conseguida seguindo uma gramática específica, ilustrada na Figura 7.

Consideremos μ uma **função de distribuição** que atribui probabilidades aos elementos de `X` **(1)**. Quando pensamos numa distribuição sobre `X`, representada como `DX`, a função μ pode ser estendida para atribuir probabilidades $\mu(x_i)$ aos elementos x_i **(2)**.

$$\textbf{(1)}\ \ \mu : X \to [0,1] \ \ \rightsquigarrow \ \ \sum_{i}^{n} p_i * x_i \qquad \textbf{(2)}\ \ \mu : X_n \to [0,1] \ \ \rightsquigarrow \ \ \sum_{i}^{n} \mu(x_i) * x_i$$

Agora, consideremos uma função `f` que mapeia elementos do conjunto `X` para `DY`. A notação f^*, por sua vez, representa uma extensão da função anterior para que esta possa operar em distribuições. Esta fórmula pode ser interpretada como a aplicação da função `f` a cada elemento x_i da distribuição `DX`, seguida da multiplicação de cada resultado pela probabilidade correspondente p_i, retornando uma nova distribuição `DY`.

$$\frac{f : X \to DY}{f^* : DX \to DY} \qquad f^*(\sum_{i}^{n} p_i * x_i) = \sum_{i}^{n} p_i * f(x_i)$$

O conhecimento destas fórmulas é extremamente útil para entender o comportamento de funções e de distribuições em conjuntos. Com isto, podemos prosseguir para a construção da semântica da nossa linguagem, com base numa gramática que inclui a atribuição de valores a variáveis, a sequência de programas, condições e ciclos *while*. Adicionalmente, temos a regra $p +_p q$ exclusiva à nossa linguagem, que executa o comando `p` com probabilidade `p` e o comando `q` com probabilidade `1-p`.

```
Prog(X) ∋ x := t | p +p q | p ; q | if b then p else q | while b do {p}
```

A semântica foi implementada através do mónada de probabilidades e da linguagem *Haskell*, aproveitando o código fornecido nas aulas. A ideia por trás de cada regra é:

- No caso da **atribuição**, a distribuição representativa de atribuir um valor a uma variável é trivial, i.e., representa 100% de certeza.
- No caso da **sequência**, primeiro avalia-se o programa `p` e obtém-se uma nova distribuição de probabilidade. A seguir, avalia-se o programa `q` em cada um dos estados σ_i derivados da execução anterior, obtendo uma distribuição μ_i correspondente a cada avaliação. Por fim, essas duas distribuições são combinadas, multiplicando as probabilidades p_i pelo resultado do programa `q` em cada estado.
- No caso da **junção**, o programa `p` é executado com a probabilidade `p` e o programa `q` é executado com a probabilidade `(1-p)`. O resultado final é a combinação das duas distribuições obtidas ao avaliar os programas `p` e `q`.

- No caso da **condição *if***, o programa `p` é executado se a condição `b` for verdadeira; caso contrário, a alternativa é o programa `q`. Ambos são executados com a probabilidade `p`. O resultado final é a distribuição obtida pelo ramo escolhido.
- No caso do **ciclo *while***, o programa `p` é executado repetidamente enquanto a condição `b` for verdadeira, retornando uma nova distribuição para cada iteração. Caso contrário, o ciclo é encerrado, sendo devolvido o estado atual cuja distribuição é trivial. No fim, combinam-se todas as distribuições resultantes, multiplicando as probabilidades p_i pela distribuição μ_i em cada estado.

$$\frac{\langle \mathtt{t}, \sigma\rangle \Downarrow r}{\langle \mathtt{x := t}, \sigma\rangle \Downarrow 1 \cdot \sigma[r/x]} \ (asg)$$

$$\frac{\langle \mathtt{p}, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sigma_i \qquad \forall i \leq n. \langle \mathtt{q}, \sigma_i\rangle \Downarrow \mu_i}{\langle \mathtt{p\ ;\ q}, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \mu_i} \ (seq) \qquad \frac{\langle \mathtt{p}, \sigma\rangle \Downarrow \mu_p \qquad \langle \mathtt{q}, \sigma\rangle \Downarrow \mu_q}{\langle \mathtt{p}\ +_p\ \mathtt{q}, \sigma\rangle \Downarrow p \cdot \mu_p + (1-p) \cdot \mu_q} \ (sum)$$

$$\frac{\langle \mathtt{b}, \sigma\rangle \Downarrow \mathtt{tt} \qquad \langle \mathtt{p}, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sigma_i}{\langle \mathtt{if\ b\ then\ p\ else\ q}, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sigma_i} \ (if1) \qquad \frac{\langle \mathtt{b}, \sigma\rangle \Downarrow \mathtt{ff} \qquad \langle \mathtt{q}, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sigma_i}{\langle \mathtt{if\ b\ then\ p\ else\ q}, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sigma_i} \ (if2)$$

$$\frac{\langle \mathtt{b}, \sigma\rangle \Downarrow \mathtt{tt} \qquad \langle \mathtt{p}, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sigma_i \qquad \forall i \leq n. \langle \mathtt{while\ b\ do\ p}, \sigma_i\rangle \Downarrow \mu_i}{\langle \mathtt{while\ b\ do\ \{p\}}, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \mu_i} \ (wh1)$$

$$\frac{\langle \mathtt{b}, \sigma\rangle \Downarrow \mathtt{ff}}{\langle \mathtt{while\ b\ do\ \{p\}}, \sigma\rangle \Downarrow 1 \cdot \sigma} \ (wh2)$$

Figura 7: Regras de semântica da linguagem probabilística

3.2 Implementação em Haskell

De modo a mapear as regras de semântica em instruções compreensíveis pelo computador, o grupo desenvolveu a função `semantic`. Essa função aceita um argumento do tipo `WhileTerm` e outro do tipo `Memory`, e retorna uma distribuição do tipo `Dist Double`.

```
semantic :: WhileTerm -> Memory -> Dist Memory
```

Os dados do tipo `WhileTerm` podem ser, por sua vez, definidos da seguinte forma:

```
data WhileTerm = Asg Var LinearTerm
    | Sum Float WhileTerm WhileTerm
    | Seq WhileTerm WhileTerm
    | Ife BooleanTerm WhileTerm WhileTerm
    | Whi BooleanTerm WhileTerm deriving Show
```

Adicionalmente, temos a função `por` que recebe dois programas `a` e `b`. Na situação desta optar pelo primeiro ramo, ele é executado com a probabilidade `p`. Caso contrário, executa o segundo com a probabilidade de `1-p`.

```
por :: ProbRep -> (Dist a, Dist a) -> Dist a
por p (x, y) = do
  a <- x
  b <- y
  choose p a b
```

Comecemos pelo caso em que a função `semantic` recebe um comando que atribui um valor a uma variável. Primeiramente, através da função `asgMem`, substitui-se o valor da variável `x` pelo valor `r`, que é o valor de `t` em memória. A seguir, como esta distribuição é trivial, ditamos que a probabilidade deste comando é `1.0`.

```
asgMem :: Var -> Double -> Memory -> Memory
asgMem x d mem = \v -> if v == x then d else mem v

semantic (Asg x t) mem = do
                         let r = semLinear t mem
                         return asgMem x r mem
```

O comando `Sum`, por outro lado, dita que o programa `p` é executado com a probabilidade `prob` (enunciado no excerto de código abaixo) e o programa q é executado com a probabilidade `(1-prob)`. Isto feito com o auxílio da função `por`.

```
semantic (Sum prob p q) mem = do
                                resultP <- semantic p mem
                                resultQ <- semantic q mem
                                por prob (return resultP, return resultQ)
```

Passando para o comando de sequência, a ideia imposta é executar o primeiro programa e depois, conforme o resultado `r` obtido com essa execução, é executado o programa `q` com a memória atualizada com este resultado.

```
semantic (Seq p q) mem = do
                         r <- semantic p mem
                         semantic q r
```

Se a função `semantic` receber, no entanto, um termo `WhileTerm` do tipo `Ife`, então recorre-se a função `boolTerm` para determinar se a condição é verdadeira ou falsa. Sendo verdadeira, o programa `p` é executado; caso contrário, é executado o programa `q`.

```
semantic (Ife b p q) mem = do
                           let condition = boolTerm b mem
                           if condition
                             then semantic p mem
                             else semantic q mem
```

Finalmente, a última possibilidade é receber um ciclo *while*, onde, mais uma vez, se a função `boolTerm` verificar que a condição é verdadeira, executa-se o programa `p`, efetuando uma nova iteração do ciclo com a memória atualizada. Caso contrário, a memória mantém-se igual.

```
semantic (Whi b p) mem = do
                           let condition = boolTerm b mem
                           if condition
                             then do
                               r <- semantic p mem
                               semantic (Whi b p) r
                           else return mem
```

3.3 Registo de mensagens

Como aspeto adicional, o grupo extendou as regras de semântica parar cobrirem o registo das mensagens com os resultados que vão sendo obtidos. No entanto, não conseguimos traduzir as regras na Figura 8 em instruções compreendidas pelo computador.

Sempre que ocorre a atribuição de um valor a uma variável, é registada uma nova mensagem ao conjunto. Relativamente às restantes regras, cada execução retorna um conjunto de mensagens que transportam os seus resultados. A única exceção é a regra `wh2` que, como a condição é falsa, retornará uma lista de mensagens vazia.

$$\frac{\langle \mathtt{t}, \sigma\rangle \Downarrow n, r}{\langle \mathtt{x := t}, \sigma\rangle \Downarrow \mathtt{m} \ +\!\!+ \ \mathtt{n}, 1 \cdot \sigma[r/x]} \ (asg)$$

$$\frac{\langle \mathtt{p}, \sigma\rangle \Downarrow m, \sum_i^n p_i \cdot \sigma_i \qquad \forall i \leq n. \langle \mathtt{q}, \sigma_i\rangle \Downarrow n, \mu_i}{\langle \mathtt{p \ ; \ q}, \sigma\rangle \Downarrow \mathtt{m} \ +\!\!+ \ \mathtt{n}, \sum_i^n p_i \cdot \mu_i} \ (seq)$$

$$\frac{\langle \mathtt{p}, \sigma\rangle \Downarrow m, \mu_p \qquad \langle \mathtt{q}, \sigma\rangle \Downarrow n, \mu_q}{\langle \mathtt{p} \ +_p \ \mathtt{q}, \sigma\rangle \Downarrow \mathtt{m} \ +\!\!+ \ \mathtt{n}, p \cdot \mu_p + (1-p) \cdot \mu_q} \ (sum)$$

$$\frac{\langle \mathtt{b}, \sigma\rangle \Downarrow \mathtt{tt} \qquad \langle \mathtt{p}, \sigma\rangle \Downarrow m, \sum_i^n p_i \cdot \sigma_i}{\langle \mathtt{if \ b \ then \ p \ else \ q}, \sigma\rangle \Downarrow m, \sum_i^n p_i \cdot \sigma_i} \ (if1) \qquad \frac{\langle \mathtt{b}, \sigma\rangle \Downarrow \mathtt{ff} \qquad \langle \mathtt{q}, \sigma\rangle \Downarrow m, \sum_i^n p_i \cdot \sigma_i}{\langle \mathtt{if \ b \ then \ p \ else \ q}, \sigma\rangle \Downarrow m, \sum_i^n p_i \cdot \sigma_i} \ (if2)$$

$$\frac{\langle \mathtt{b}, \sigma\rangle \Downarrow \mathtt{tt} \qquad \langle \mathtt{p}, \sigma\rangle \Downarrow m, \sum_i^n p_i \cdot \sigma_i \qquad \forall i \leq n. \langle \mathtt{while \ b \ do \ p}, \sigma_i\rangle \Downarrow n, \mu_i}{\langle \mathtt{while \ b \ do \ \{p\}}, \sigma\rangle \Downarrow \mathtt{m} \ +\!\!+ \ \mathtt{n}, \sum_i^n p_i \cdot \mu_i} \ (wh1)$$

$$\frac{\langle \mathtt{b}, \sigma\rangle \Downarrow \mathtt{ff}}{\langle \mathtt{while \ b \ do \ \{p\}}, \sigma\rangle \Downarrow [], 1 \cdot \sigma} \ (wh2)$$

Figura 8: Regras de semântica extendida com o registo de mensagens

Para isto, nós precisaríamos de recorrer à função `mwrite`, facultada no ficheiro `LogMessages.hs`. Ela recebe uma *string* com o resultado e uma variável do tipo `LogList a`, definida como `Log [(String, a)]`. O objetivo dessa função é adicionar a mensagem (***string***) ao conjunto já existente (**LogList a**).

```
mwrite :: String -> LogList a -> LogList a
mwrite msg l = Log $ let l' = remLog l in map (\(s,x) -> (s ++ msg, x)) l'
```

Referências

[1] Xintao Ma, Jonas Rinast, Sibylle Schupp, and Dieter Gollmann. Evaluating On-line Model Checking in UPPAAL-SMC using a Laser Tracheotomy Case Study. In Volker Turau, Marta Kwiatkowska, Rahul Mangharam, and Christoph Weyer, editors, *5th Workshop on Medical Cyber-Physical Systems*, volume 36 of *OpenAccess Series in Informatics (OASIcs)*, pages 100–112, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.