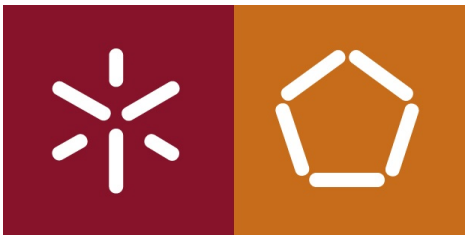

Universidade do Minho
Mestrado em Engenharia Informática
1ºano - 2º Semestre

Programação Cíber-física

Projeto Final

A85635 - André Nunes
PG50419 - Hugo Fernandes

Junho, 2023

Conteúdo

Pergunta 1

Modelo

A fim de implementar o cenário proposto na primeira pergunta foram desenvolvidos dois modelos: o *Plane* e o *Controller*, como vistos na seguinte figura.

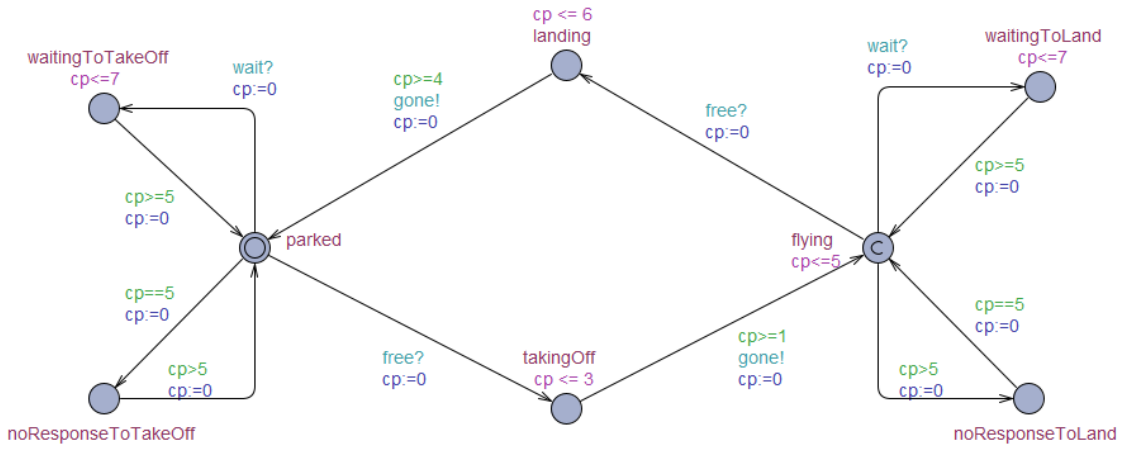

Figura 1: Modelo do *Plane*

Neste modelo do *Plane* são tratados os requisitos pedido no enunciado. Os estados *parked* e *flying* representam as posições básicas de um avião. *Flying* é um estado *committed*, para garantir que é abandonado, uma vez que um avião tem, eventualmente, de aterrar. Por outro lado, os estados *landing* e *takingOff*, representam o uso de uma mesma pista, quer para aterrar, quer para descolar. Temos ainda estados auxiliares que servem para gerir a espera pela pista, seja esta espera por indicação(*waitingToTakeOff* / *WaintingToLand*) ou por falta de resposta do *Controller*(*noResponseToTakeOff* / *noResponseToLand*).

Os aviões devem pedir permissão para usar a pista através de *free?* e após a usarem devem notificar com *gone!*.

Quando estão à espera de lugar na pista, devem, se não houver lugar, receber *wait?*. No entanto, se essa notificação não chegar, deve ocorrer uma espera diferente (*noResponseToTakeOff* / *noResponseToLand*).

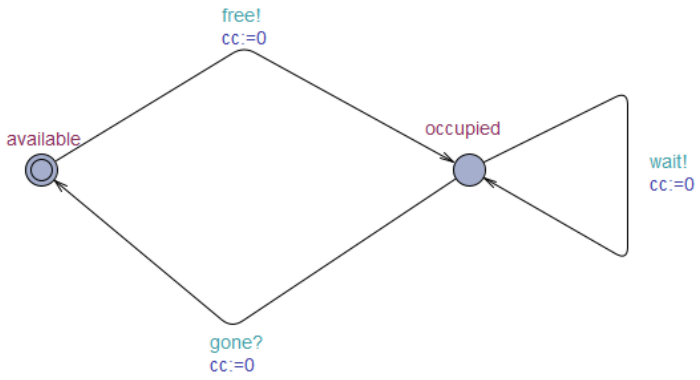

Figura 2: Modelo do *Controller*

Quanto ao *Controller*, tal como o nome indica controla o estado da pista, que pode ser *available* ou *occupied*. O *Controller* deve notificar os interessados que a pista está livre com *free!*, momento em que passa a estar ocupada. De igual forma deve receber a informação de que a pista voltou a estar livre com *gone?*. Quando a pista está ocupada deve ainda notificar os interessados com *wait!*.

Propriedades

Foram também definidas algumas propriedades para testar a correção do modelo.

Propriedade 1

Esta propriedade visa garantir que, se a pista estiver livre(*Controller* em *available*) nenhum avião pode estar a descolar ou a aterrar.

$$\mathbf{A\Box}\ \mathbf{Process3.available}\ \mathbf{imply}\ \mathbf{not}(\mathbf{Process1.takingOff}\ \&\&\ \mathbf{Process1.landing}\ \&\&\ \mathbf{Process2.takingOff}\ \&\&\ \mathbf{Process2.landing})$$

Propriedade 2

Esta propriedade visa garantir que, se um avião está num estado, não pode estar em outro estado.

$$\begin{aligned}\mathbf{A\Box Process1.parked}\ \mathbf{imply}\ \mathbf{not}(&\mathbf{Process1.flying}\ \&\&\mathbf{Process1.landing}\\ &\&\&\mathbf{Process1.takingOff}\ \&\&\mathbf{Process1.waitingToTakeOff}\\ &\&\&\mathbf{Process1.noResponseToTakeOff}\ \&\&\mathbf{Process1.waitingToLand}\\ &\&\&\mathbf{Process1.noResponseToLand})\end{aligned}$$

Propriedade 3

Esta propriedade visa garantir que se um avião está a voar, terá eventualmente que aterrar.

$$\mathbf{Process1.flying} \rightsquigarrow \mathbf{Process1.landing}$$

Propriedade 4

Esta propriedade visa garantir que se um avião está a aterrar, eventualmente ficará estacionado.

$$\mathbf{Process1.landing} \rightsquigarrow \mathbf{Process1.parked}$$

Propriedade 5

Esta propriedade visa garantir que apenas um avião pode usar a pista de cada vez.

$$\begin{aligned}&\mathbf{A\Box}(\mathbf{Process1.landing\ or\ Process1.takingOff})\\ &\quad\mathbf{imply}\ \mathbf{not}(\mathbf{Process2.landing\ or\ Process2.takingOff})\end{aligned}$$

Propriedade 6

Esta propriedade visa garantir que se a pista está ocupada, eventualmente terá de voltar a ficar livre.

$$\textbf{\textit{Process3.occupied}} \rightsquigarrow \textbf{\textit{Process3.available}}$$

Propriedade 7

Esta propriedade visa garantir que não existem deadlocks no modelo.

$$\textbf{\textit{A}}\Box\ \textbf{\textit{not deadlock}}$$

Propriedade 8

Esta propriedade visa garantir que, a pista poderá, eventualmente, tornar-se ocupada.

$$\textbf{\textit{E}}\Diamond\textbf{\textit{Process3.occupied}}$$

Pergunta 2

Diferenças entre modelação e verificação

No mundo da informática e da engenharia de software, os processos de modelação e verificação desempenham papéis cruciais para garantir a exatidão e a eficácia dos sistemas de software. Através da modelação são criadas representações abstratas dos sistemas para captar os seus aspetos e comportamentos essenciais. A verificação, por outro lado, envolve analisar e verificar se um sistema satisfaz o seu comportamento desejado e está em conformidade com os requisitos especificados. Estes dois conceitos são fundamentais para a conceção e análise de sistemas, fornecendo uma base sólida para o desenvolvimento de sistemas de software fiáveis e eficientes.

Objetivos

O objetivo da modelação e da verificação no desenvolvimento de software é garantir a fiabilidade, a correção e a eficácia dos sistemas de software. A modelação permite a criação de representações abstratas que captam os aspetos e comportamentos essenciais de um sistema, facilitando a compreensão, a comunicação e a análise. A verificação, por outro lado, tem como objetivo validar se um sistema cumpre os requisitos especificados, adere às propriedades desejadas e funciona como pretendido. Estas atividades desempenham um papel fundamental nas fases iniciais do desenvolvimento, lançando as bases para uma implementação bem sucedida do software.

A modelação envolve a criação de representações abstratas de um sistema, capturando os seus aspectos essenciais, estrutura e comportamento. Os modelos podem ser gráficos ou textuais e são utilizados para facilitar a compreensão, a comunicação e a análise das propriedades de um sistema.

O objetivo da verificação é garantir que um sistema ou software cumpre os requisitos especificados ou respeita determinadas propriedades. Implica analisar e verificar se um sistema satisfaz o comportamento desejado, as restrições de segurança e a consistência lógica.

Nível de abstração

A modelação e a verificação operam em diferentes níveis de abstração, proporcionando perspectivas únicas sobre os sistemas de software. A modelação envolve a captura da estrutura geral e do comportamento de um sistema sem se aprofundar em pormenores de implementação de baixo nível. Permite que os projetistas se concentrem nos aspetos concetuais, realçando as características e relações essenciais do sistema. Por outro lado, a verificação opera a um nível mais detalhado, examinando a implementação para garantir que está alinhada com o comportamento pretendido especificado nos modelos. Envolve uma análise meticulosa, verificando propriedades e requisitos específicos do sistema.

A modelação funciona a um nível mais elevado de abstração, captando a estrutura e o comportamento globais de um sistema sem entrar em detalhes de implementação de baixo nível. Permite que os desenvolvedores se concentrem nos aspectos conceptuais e exprimam as características essenciais do sistema.

A verificação funciona a um nível mais pormenorizado, centrando-se em propriedades e requisitos específicos. Envolve o exame minucioso da implementação do sistema para validar se está em conformidade com o comportamento pretendido, tal como especificado no modelo.

Ferramentas e técnicas

A modelação e a verificação dependem de uma série de ferramentas e técnicas para captar eficazmente o comportamento do sistema e garantir a sua correção. Embora existam várias opções disponíveis, no decorrer do perfil de MFP, ao qual ingressamos, tivemos a oportunidade de aprender e utilizar algumas dessas técnicas e ferramentas.

O Uppaal é uma ferramenta popular utilizada para modelação e verificação de sistemas em tempo real. Permite a criação de modelos de autómatos temporizados e suporta a verificação formal através da verificação de modelos, simulação e análise de acessibilidade. O Uppaal fornece uma interface visual para projetar e analisar o comportamento do sistema, tornando-o particularmente útil para aplicações de tempo crítico.

O Frama-C é uma estrutura de análise de software projetada especificamente para a verificação de programas em C. Ele incorpora várias técnicas de análise estática, como interpretação abstrata e prova formal, para detetar erros, vulnerabilidades e comportamentos indefinidos no código C. O Frama-C suporta a verificação formal usando técnicas dedutivas, o que o torna valioso para garantir a correção de sistemas de software baseados em C.

TLA+ (Temporal Logic of Actions) é uma linguagem de especificação formal e um conjunto de ferramentas de verificação de modelos. Permite a descrição de comportamentos, propriedades e restrições do sistema utilizando uma linguagem de alto nível. O TLA+ fornece um poderoso verificador de modelos que analisa exaustivamente os modelos do sistema para detetar erros, consistência e outras propriedades. Ele é conhecido pela sua capacidade de averiguar a correção de sistemas concorrentes e distribuídos.

Alloy é uma linguagem de modelagem formal leve e uma ferramenta de análise. É particularmente adequada para modelar e analisar sistemas de software com estruturas e relacionamentos complexos. O Alloy usa uma linguagem de especificação baseada em lógica de primeira ordem e emprega um "solver"baseado em SAT para verificação de modelos. Permite a deteção de falhas de conceção, inconsistências e contra-exemplos em modelos de sistemas, ajudando a aperfeiçoar e melhorar as concepções de software.

Estas ferramentas exemplificam a gama diversificada de opções disponíveis para efeitos de modelação e verificação. Ao utilizar estas ferramentas, os desenvolvedores de software podem analisar eficazmente o comportamento do sistema, detetar potenciais problemas e validar a correção e a fiabilidade dos seus sistemas de software.

Programação

A programação refere-se ao processo de escrever código utilizando uma linguagem de programação específica para implementar um sistema baseado nos modelos e requisitos especificados nas fases anteriores. Implica traduzir as representações abstractas e as decisões de conceção em instruções executáveis que um computador possa compreender e executar. A programação exige atenção aos pormenores, pensamento algorítmico e conhecimento dos paradigmas de programação, da sintaxe e das melhores práticas.

Natureza complementar Embora a modelação, a verificação e a programação sejam atividades distintas, estão estreitamente interligadas e

complementam-se mutuamente ao longo do ciclo de vida do desenvolvimento de software.

Processo iterativo A modelação e a verificação andam muitas vezes de mãos dadas, alternando entre si. Os modelos fornecem uma base para a verificação, permitindo que os analistas raciocinem sobre as propriedades do sistema e identifiquem potenciais problemas numa fase inicial. A verificação, por sua vez, ajuda a validar a correção dos modelos e a aperfeiçoá-los ainda mais. Este processo iterativo melhora a qualidade geral da conceção do sistema.

Melhoria da compreensão e da comunicação A modelação melhora a comunicação e a compreensão entre as partes interessadas, uma vez que fornece uma representação visual ou textual que pode ser facilmente compreendida por várias pessoas envolvidas no processo de desenvolvimento. A verificação atua como um meio de garantir que o modelo representa com precisão o comportamento desejado do sistema, conduzindo a uma compreensão partilhada e reduzindo a ambiguidade.

Deteção e correção de erros As técnicas de verificação, como a verificação e o teste de modelos, ajudam a identificar erros ou inconsistências no sistema implementado. Estas técnicas podem revelar potenciais problemas não detetados durante a fase de modelação, ajudando a descobrir e a corrigir defeitos antes da implantação. A programação faz a ponte entre os modelos abstractos e as implementações concretas, permitindo a deteção e a resolução de discrepâncias entre a conceção prevista e a implementação real.

Refinamento e otimização A programação permite a realização de características e funcionalidades complexas do sistema. À medida que os modelos são traduzidos em código, os programadores têm a oportunidade de otimizar e aperfeiçoar o desempenho, a eficiência e a capacidade de manutenção do sistema. Durante esta fase, podem ser utilizadas técnicas de verificação para garantir que o código optimizado continua a respeitar as propriedades desejadas.

Conclusão

Em conclusão, a modelação, a verificação e a programação são componentes essenciais do processo de desenvolvimento de software. Enquanto a modelação e a verificação se centram na conceção e análise de sistemas de alto nível, a programação dá vida a essas concepções. Estas atividades são interdependentes e reforçam-se mutuamente, com a modelação e a verificação a ajudarem na criação de sistemas corretos e fiáveis, e a programação a implementar as conceções previstas e a facilitar o aperfeiçoamento contínuo. Ao compreender as diferenças e reconhecer a natureza complementar destas atividades, os programadores de software podem criar sistemas de software robustos, eficientes e bem validados.

Pergunta 3

No âmbito da terceira implementamos uma linguagem probablistica em Haskell partindo da linguagem while desenvolvida no TPC2 desta unidade curricular e derivamos a sua semântica.

Semântica

Para tal começamos por derivar a sua semântica apartir da regra da composição sequencial que nos foi fornecida no enunciado. Temos portanto a nossa primeira regra semântica.

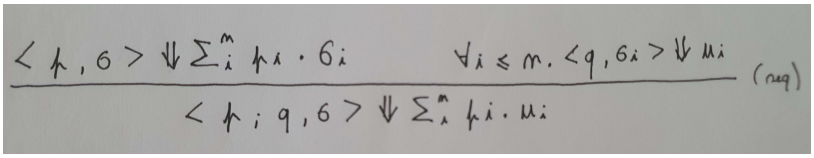

Figura 3: Seq

Esta regra diz que tendo um estado de memória e a sua distribuição de probabilidade vão sendo executados sequencialmente os programas nessa memória e nos consequentes estado de memória possíveis relativos a uma distribuição de propabilidade após a execução de cada programa. A cada iteração os estados possíveis afetam os estados existentes, criando novos estados que são passados para a próxima iteração criando ainda mais estados.

De seguida, começamos por derivar a regra do (Assignment), esta regra permanece inalterada relativamente à linguagem While uma vez que embora existam vários estados de memória esta simplesmente altera os possíveis estados de memória atual.

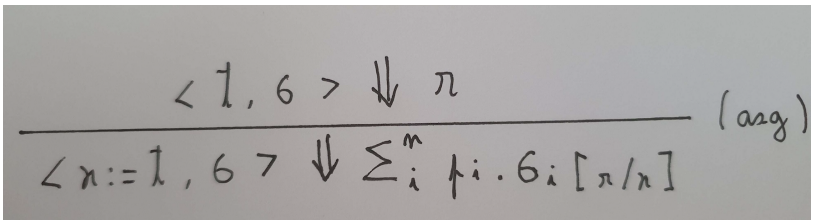

Figura 4: Asg

Relativamente à regra do (If...Then...Else...) foi necessário dividir em 2 casos as regras da semântica uma vez que caso a condição seja verdade, um programa é executado e caso seja falsa outro é executado na sua vez alterando o conjunto de memórias possíveis.

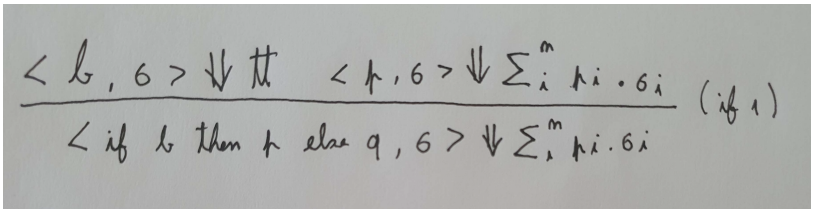

Figura 5: If1

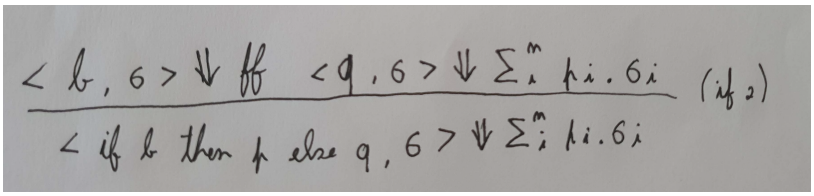

Figura 6: If2

Postriormente derivamos a regra do (While) onde foi também necessário dividir em 2 regras semânticas quando a condicional se verifica ou não. No caso da condicional se verificar o programa vai ser executado alterando os estados de memória possíveis de forma sequencial sempre que esta se verifica. A cada iteração os estados possíveis afetam os estados existentes, criando novos estados que são passados para a próxima iteração criando ainda mais estados.

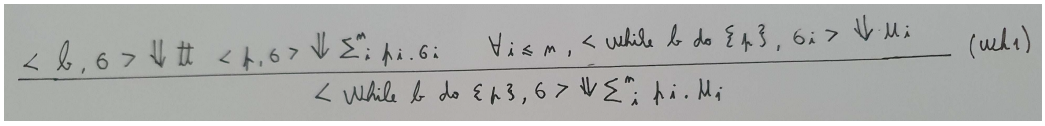

Figura 7: Wh1

No caso da condicional não se verificar a memória é inalterada uma vez que nenhum programa acaba por ser executado após a verificação da condicional.

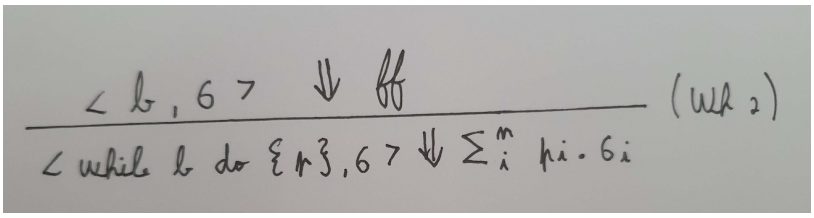

Figura 8: Wh2

Finalmente derivamos então a regra da escolha probabilistica pedida no enunciado. Esta regra afirma que dados 2 programas (a,b) e uma distribuição de probabilidade (p), o programa (a) é executado com probabilidade (p) e o programa (b) é executado com probabilidade (1-p).

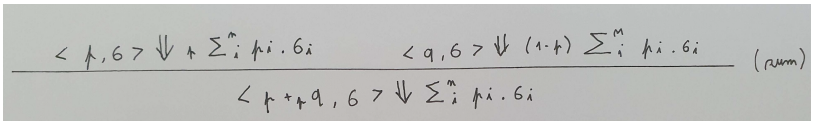

Figura 9: Sum1

Como podemos perceber pelas regras de semântica acima, após a execução dos programas com a probabilidade (p) respetiva os estados de memória possíveis são alterados de acordo com os programas executados e a distribuição probabilística associada.

Implementação

Após toda a definição destas regras implemantamos esta pequena linguagem probabilistica em Haskell recorrendo à Monad probabilistica disponíbilizada pelos professores

nas aulas. Para tal adaptamos a linguagem desenvolvida no TPC2 e simplesmente adaptamos as alterações efetuadas de acordo com as regras de semântica derivadas.

Alterações

O código-fonte sofreu várias modificações para incorporar programas de escolha probabilística e registar os estados da memória. Foram efectuadas as seguintes alterações:

Primeiramente um novo construtor chamado ProbChoice foi adicionado ao tipo de dados "WhP". Este construtor representa um programa de escolha probabilística e recebe três argumentos: uma distribuição de probabilidades "ProbRep"e dois programas while "WhP"que representam as escolhas.

A função "wsem", responsável pela execução de programas while, foi modificada para lidar com o construtor "ProbChoice". Ao encontrar um programa "ProbChoice", a função usa a função "choose"para selecionar probabilisticamente uma das duas escolhas com base na distribuição de probabilidades fornecida. A escolha selecionada é então passada para uma chamada recursiva a wsem para continuar a execução.

A assinatura da função "wsem"foi atualizada para incluir um parâmetro adicional chamado "states", que é uma lista de estados de memória encontrados durante a execução do programa. Esta lista é utilizada para manter o registo dos estados de memória em diferentes pontos do programa.

No caso "Asg", que representa a atribuição de variáveis, o estado de memória atualizado é acrescentado à lista de estados utilizando a função "chMem". Isso garante que o novo estado de memória seja adicionado à lista de estados.

No caso "Seq", após a execução do primeiro programa e a obtenção dos estados de memória resultantes (estados'), a lista de estados é passada como argumento para a execução do segundo programa (q). Isto permite a execução sequencial de vários programas e a acumulação de estados de memória.

No geral, essas mudanças permitem a execução de programas while com escolha probabilística e o controle de estados de memória. A função wsem agora lida com uma gama mais ampla de estruturas, permitindo uma execução de programas mais expressivos e flexíveis.