Métodos Formais em Engenharia de Software
Teste, 16 de Dezembro de 2023

Número: ___________ Nome: ____

1. SAT *(3.5 pontos)*

A Ana, o Bernardo, o Carlos, o Daniel, a Emilia, o Francisco e a Gabriela, moram na mesma casa. Mas a casa tem as seguintes regras de funcionamento.

1. Se a Ana estiver em casa, então o Bernardo também tem de estar.
2. O Daniel ou a Emilia, ou ambos, estão em casa.
3. O Bernardo ou o Francisco estão em casa, mas não ambos.
4. O Daniel e o Carlos estão ambos em casa ou ambos fora de casa.
5. Se a Emilia estiver em casa, então a Ana e o Daniel também estão.
6. A Emilia e o Francisco não podem estar sozinhos em casa.
7. A Ana nunca se ausenta de casa, excepto se o Francisco e o Carlos estiverem os dois em casa.

a) Codifique este problema em lógica proposicional. Assinale o que denota cada variável proposicional que introduzir, e escreva um conjunto de fórmulas proposicionais adequado à sua modelação.

b) Diga, justificando, como poderia usar um SAT solver para indagar sobre a veracidade das seguintes afirmações:

(1) O Daniel e o Carlos estão sempre em casa.
(2) Não será possível ter todos os moradores simultaneamente em casa.
(3) O Bernardo e o Francisco nunca se vão encontrar em casa.

Resposta:

a) Vamos usar a inicial do nome de cada pessoa para denotar que ela está em casa.

1. $A \to B$
2. $D \lor E$
3. $(B \lor F) \land \neg (B \land F)$
4. $(D \land C) \lor (\neg D \land \neg C)$
5. $E \to (A \land D)$
6. $(E \land F) \to (A \lor B \lor C \lor D \lor F)$
7. $\neg (F \land C) \to A$

b) Seja Γ o conjunto de fórmulas definidas na alínea anterior.

1. A frase é verdadeira se $\Gamma \models D \land C$
2. A frase é verdadeira se $\Gamma \models \neg (A \land B \land C \land D \land E \land F \land F)$
3. A frase é verdadeira se $\Gamma \models \neg (B \land F)$

Sabemos que para qualquer fórmula ϕ, $\Gamma \models \phi$ sse $\Gamma, \neg\phi$ UNSAT. Com base neste teorema, acrescentamos ao conjunto Γ a negação da fórmula que representa a propriedade que queremos verificar e conferimos se a resposta do solver é UNSAT.

Métodos Formais em Engenharia de Software
Teste, 16 de Dezembro de 2023

Número: __________ Nome: ____

2. SMT *(2.5 pontos)*

Considere o seguinte programa C sobre inteiros:

```
x = y + z;
if (x < z) {
    x = x + y;
    y = 2 * y;
}
else
    x = x - y;
```

a) Faça a codificação lógica deste programa na teoria de inteiros.
b) Diga, justificando, como poderia usar um SMT *solver* para verificar se este programa satisfaz a seguinte propriedade: "*Se o valor inicial de y é positivo, então o seu valor não é alterado*". (Não precisa de usar a sintaxe específica do SMT *solver*).

Resposta:

a) A codificação é feita pelo seguinte conjunto de fórmulas:

$x1 = y0 + z0$
$x1 < z0 \rightarrow x2 = x1 + y0$
$x1 < z0 \rightarrow y1 = 2 * y0$
$\neg (x1 < z0) \rightarrow x3 = x1 - y0$
$x4 = x1 < z0\ ?\ x2 : x3$
$y2 = x1 < z0\ ?\ y1 : y0$

b) A propriedade que queremos verificar é $y0 > 0 \rightarrow y2 = y0$
Para isso, acrescentamos a sua negação $\neg (y0 > 0 \rightarrow y2 = y0)$ ao SMT-solver e se o novo conjunto e fórmulas for UNSAT é porque a propriedade se verifica.

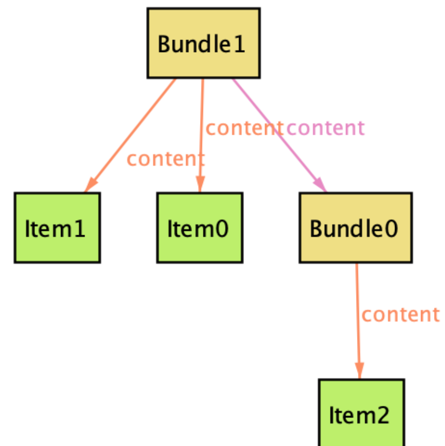
Métodos Formais em Engenharia de Software
Teste, 16 de Dezembro de 2023

Número: __________ Nome: ____

3. Modelação estrutural com Alloy *(4 pontos)*

Considere o seguinte modelo de uma loja online, onde existem vários produtos (*items*) à venda, podendo esses produtos ser agrupados em pacotes promocionais (*bundles*), cujos produtos devem ser adquiridos em conjunto. Os pacotes podem ser constituídos por outros pacotes, tal como se vê na figura à direita. Cada utilizador (*user*) tem um carrinho de compras (*cart*), onde deposita os produtos que deseja comprar. Quando faz uma encomenda, todos os produtos no carrinho, se ainda estiverem disponíveis (em *stock*), passam para o estado pendente (*pending*), e quando um produto é despachado passa para o estado entregue (*delivered*).

```
sig Item {}
sig stock in Item {}
sig Bundle {
      content : set Item + Bundle
}
sig User {
      pending : set Item,
      delivered : set Item,
      cart : set Item
}
```



Especifique as seguintes propriedades deste sistema:

a) Cada produto só pode ser adquirido por um utilizador. Um produto considera-se adquirido se estiver pendente ou entregue.
b) Todos os pacotes têm que ter pelo menos um produto.
c) Os produtos em *stock* são os que não estão adquiridos.
d) Se um produto de um pacote estiver no carrinho de um utilizador, todos os restantes produtos agrupados também têm que lá estar. Por exemplo, no caso ilustrado na figura, se o `Item0` estiver no carrinho de um utilizador então o `Item1` e o `Item2` também têm que lá estar.

Resposta:

Há muitas especificações possíveis para estas propriedades. Uma possível solução é a seguinte:

```
a) all i : Item | lone (pending+delivered).i
b) all b : Bundle | some b.content & Item
c) stock = Item - User.(pending+delivered)
d) all u : User, i : u.cart, b : ^content.i | b.^content & Item in u.cart
```

Métodos Formais em Engenharia de Software
Teste, 16 de Dezembro de 2023

Número: __________ Nome: ____

4. Modelação comportamental com Alloy *(4 pontos)*

Considere o seguinte *concept* de carrinho de compras, baseado numa versão simplificada do modelo anterior, onde não temos pacotes de produtos, e onde o conjunto *stock* e as relações binárias *cart*, *pending* e *delivered* são agora mutáveis. Neste concept temos as seguintes acções: adicionar um produto ao carrinho (*add*); remover um produto do carrinho (*remove*); encomendar todos os produtos no carrinho de um utilizador, caso ainda estejam em stock (*order*); cancelar a encomenda de um produto (*cancel*); fazer a entrega de um produto previamente encomendado (*deliver*).

```
sig Item {}
var sig stock in Item {}
sig User {
      var pending : set Item,
      var delivered : set Item,
      var cart : set Item
}
```

```
pred add [u:User, i:Item] { ... }
pred remove [u:User, i:Item] { ... }
pred order [u:User] { ... }
pred cancel [u:User, i:Item] { ... }
pred deliver [i:Item] { ... }
```

a) Especifique a ação `order`.
b) Especifique os seguintes princípios operacionais:
    (1) Se um utilizador cancelar a encomenda de um produto é porque antes fez uma encomenda.
    (2) Depois de entregue um produto irá sempre pertencer ao mesmo utilizador.

Resposta:

Há várias especificações possíveis para a ação e para as propriedades. Uma possível solução é a seguinte:

a)

```
pred order [u:User] {
 some u.cart                  // tem que haver alguma coisa no carrinho
 u.cart in stock              // os produtos tem que estar em stock
 cart' = cart - u->Item             // os produtos saem do carrinho
 pending' = pending + u->u.cart     // e passam a estar pendentes
 stock' = stock - u.cart            // e também deixam de estar em stock
 delivered' = delivered             // os produtos entregues ficam iguais
}
```

b)

```
   (1)   all u : User, i : Item | always (cancel[u,i] implies once order[u])
   (2)   all i : Item | always (deliver[i] implies
                              after some u : User | always i in u.delivered)
```

Métodos Formais em Engenharia de Software
Teste, 16 de Dezembro de 2023

Número: __________ Nome: ____

5. Why3 *(6 pontos)*

Pretende-se implementar em Why3 a acção *deliver* do *concept* anterior, cuja especificação em Alloy é a seguinte.

```
pred deliver [i:Item] {
      some pending.i
      pending' = pending - pending.i->i
      delivered' = delivered + pending.i->i
      stock' = stock
      cart' = cart
}
```

Considere que o estado se encontra implementado com conjuntos finitos “imperativos”, tal como apresentado de seguida.

```
use int.Int
use set.Fset
type item = int
type user = int
clone set.SetImp as Set with type elt = item
clone set.SetImp as Rel with type elt = (user,item)
type set = Set.set
type rel = Rel.set
val copy (r:rel) : rel ensures { result = r }

type stateT = { stock : set; cart : rel; pending : rel; delivered : rel }
  invariant { forall u v :user, i :item.
    mem (u,i) pending /\ mem (v,i) pending -> u = v }
  by { stock = Set.empty(); cart = Rel.empty();
       pending = Rel.empty(); delivered = Rel.empty() }

val state : stateT

let deliver (i : item)
  requires { ... }
  ensures { ... }
= ...
```

a) Traduza a especificação desta acção para WhyML, escrevendo pré- e pós-condições adequadas para a função `deliver`.
b) Apresente uma definição do corpo da função em WhyML.
c) Apresente os invariantes e variante de ciclo que permitam verificar com sucesso a função anterior.

Resposta:

a) Existem duas possibilidades de resposta (ambas seriam consideradas correctas).
A primeira consistem em traduzir de forma directa a especificação em Alloy:

```
requires { exists u:user . mem (u,i) state.pending }
ensures  { forall u:user . not mem (u,i) state.pending }
ensures  { forall u:user .
            mem (u,i) (old state.pending) -> mem (u,i) state.delivered }
ensures  { forall u:user, j :item . i<>j ->
            mem (u,j) (old state.pending) <-> mem (u,j) state.pending }
ensures  { forall u:user, j :item . i<>j ->
            mem (u,j) (old state.delivered) <-> mem (u,j) state.delivered }
ensures  { state.stock = old state.stock }
ensures  { state.cart = old state.cart }
```

A segunda solução é simplificada tendo em conta o invariante de tipo, que assegura que existe um único par (u,i) que passa de `state.pending` para `state.delivered`.

```
requires { exists u:user . mem (u,i) state.pending }
ensures  { exists u:user . mem (u,i) (old state.pending)
                   /\ state.pending == remove (u,i) (old state.pending)
                   /\ state.delivered == add (u,i) (old state.delivered) }
ensures  { state.stock = old state.stock }
ensures  { state.cart = old state.cart }
```

Métodos Formais em Engenharia de Software
Teste, 16 de Dezembro de 2023

Número: __________ Nome: ____

b, c) Não tendo em conta a informação do invariante de tipo, haverá que mover de `pending` para `delivered` todos os pares contendo o item `i`:

```
let deliver (i : item)  =
let aux = copy state.pending in
while not (is_empty aux) do
  variant { cardinal aux }
  invariant { forall u:user . mem (u,i) state.pending -> mem (u,i) aux }
  invariant { forall u:user . mem (u,i) (old state.pending) ->
                            mem (u,i) aux \/ mem (u,i) state.delivered }
  invariant { forall u:user, j :item . i<>j ->
              mem (u,j) (old state.pending) <-> mem (u,j) state.pending }
  invariant { forall u:user, j :item . i<>j ->
              mem (u,j) (old state.delivered) <-> mem (u,j) state.delivered }
  invariant { state.stock = old state.stock /\ state.cart = old state.cart }
  let (v,k) = Rel.choose_and_remove aux in
  if k = i then begin
    Rel.remove (v,i) state.pending ; Rel.add (v,i) state.delivered ;
  end ;
done
```

Pode-se no entanto tomar partido do invariante de tipo para optimizar esta versão, parando antecipadamente o ciclo depois de movido o (único) par contendo o item `i`. Isto permite também simplificar consideravelmente o invariante de ciclo.

```
let deliver (i : item)  =
let aux = copy state.pending in
while true do
  variant { cardinal aux }
  invariant { subset aux state.pending }
  invariant { exists u :user. mem (u,i) aux }
  invariant { state.pending == old state.pending }
  invariant { state.delivered == old state.delivered }
  let (v,k) = Rel.choose_and_remove aux in
  if k = i then begin
    Rel.remove (v,i) state.pending ; Rel.add (v,i) state.delivered ;
    break ;
  end ; done
```