2022

Parallele und Verteilte Systeme

Spezifikation in mCRL2 – Gruppe C1

DENNIS JONGEBLOED, 7010939
ABDULLATIF ZANABILI, 7014798
RAINER PEDDE, 7014109
DENNIS SEILER, 7011776

Inhaltsverzeichnis

Glossar

Multi-Aktion

Die Multi-Aktion beschreibt Aktionen, die gleichzeitig in parallellaufenden Prozessen ausgeführt werden können.

Aufgabe 2a: Sequenzielle Spezifikation

In dieser Aufgabe geht es darum einen Verkaufsautomaten zu erstellen der von 5 Cent bis 1 Euro den Wert von Münzen erkennt und bis zu einem Wert von 200 Cent diese akzeptiert und gutschreibt.
Des Weiteren soll der Automat Produkte mit Preisen gelistet haben und diese beim Erreichen bzw. Überschreiten des Guthabens zum Kauf anbieten.
Wird ein angebotenes Produkt gewählt, soll der Automat dieses ausgeben und den Preis vom Guthaben abziehen.
Solange ein Guthaben über 0 Cent vorhanden ist, soll die Möglichkeit bestehen es wieder zurückzugeben. Dabei soll immer die größtmögliche Münze ausgegeben werden, bis das Guthaben wieder 0 Cent beträgt.

Definition Coin und Product

Eine Münze ‘Coin’ kann eine 5 Cent/10 Cent/20 Cent/50 Cent/1 Euro Münze sein.

```
sort
    Coin = struct _5c | _10c | _20c | _50c | Euro;
```

Der Wert ‘value’ einer Münze ist eine Ganzzahl ‘Int’.

```
map
    value : Coin -> Int;    % the value of a coin as an integer
```

Die Münzen sind alle einen gewissen Betrag an Cent wert.

```
eqn
    value(_5c) = 5;
    value(_10c) = 10;
    value(_20c) = 20;
    value(_50c) = 50;
    value(Euro) = 100;
```

Ein Produkt ‘Product’ kann ein Tee/Kaffee/Kuchen/Apfel sein.

```
sort
    Product = struct tea | coffee | cake | apple;
```

Der Preis ‘price’ eines Produktes ist eine Ganzzahl ‘Int’.

```
map
    price : Product -> Int; % the price of a product as an integer
```

Die Produkte sind alle einen gewissen Betrag an Cent wert.

```
eqn
    price(tea) = 10;
    price(coffee) = 25;
    price(cake) = 60;
    price(apple) = 80;
```

Definition Aktionen

Es gibt 5 Ereignisse, die der Automat ausführen kann. Er kann eine Münze annehmen mit der Aktion ‘accept’ oder ausgeben mit ‘return’, ein Produkt anbieten mit ‘offer’ oder ausgeben mit ‘serve’ und das restliche Guthaben ausgeben mit der ‘returnChange-Aktion’.

```
act
   accept : Coin;        % accept a coin inserted into the machine
   return : Coin;        % returns change
   offer : Product;      % offer the possibility to order a certain product
   serve : Product;      % serve a certain product
   returnChange : Int;   % request to return the current credit as  change
```

Definition Prozesse

Der Prozess ‘VendingMachine’ ruft den Prozess ‘VM’ mit dem Startguthaben von 0 auf. Solange das Guthaben noch keine 200 Cent beträgt, akzeptiert der Automat Münzen und rechnet ihren Wert auf das Guthaben drauf.
Wenn das Guthaben gleich oder größer den Preis eines Produktes ist, kann das Produkt angeboten dann ausgegeben und anschließend sein Preis vom Guthaben abgezogen werden. Wenn das Guthaben größer 0 ist kann es als Wechselgeld ausgegeben werden. Dazu wird der Prozess ReturnChange aufgerufen.

```
proc
   VendingMachine = VM(0);

   VM(credit : Int) =
                        % accept coin as long as there are under 200c inside VM
                        % offer the products as long as there is enough credit to cover their price
                        % offer to return the change as long as there is any
      (credit < 200) -> sum c : Coin. accept(c) . VM(credit + value(c))
                        +  sum p : Product. (credit >= price(p)) -> offer(p) . serve(p) . VM(credit - price(p))
                        + (credit > 0) ->
                        returnChange(credit) . ReturnChange(credit)
   ;
```

Der Prozess ‘ReturnChange’ vergleicht ob Größte bzw. nächst größte Münze den Wert des Guthabens nicht überschreitet.
Ist dies nicht der Fall wird sie ausgegeben und ihr Wert dem Guthaben abgezogen. Dies wird wiederholt, bis das Guthaben 0 beträgt. Anschließend wird wieder die ‘VM’ mit dem Guthaben 0 aufgerufen.

```
            % return change biggest coin possible first
   ReturnChange(credit : Int) =
      (credit >= value(Euro)) ->
         return(Euro) . ReturnChange(credit - value(Euro))
      <>
      (credit >= value(_50c)) ->
         return(_50c) . ReturnChange(credit - value(_50c))
      <>
      (credit >= value(_20c)) ->
         return(_20c) . ReturnChange(credit - value(_20c))
      <>
```

```
      (credit >= value(_10c)) ->
         return(_10c) . ReturnChange(credit - value(_10c))
      <>
      (credit >= value(_5c)) ->
         return(_5c) . ReturnChange(credit - value(_5c))
      + (credit == 0) -> VM(0)
   ;
```

Beim Starten wird der Prozess ‘VendingMachine’ aufgerufen.

```
init
   VendingMachine
;
```

Aufgabe 2b: Parallele Spezifikation

Die TrafficLight Aufgabe, bekannt aus der Java-Aufgabe, haben wir auch in mCRL2 spezifiziert. Dabei geht es um ein Ampelsystem für eine Straßenkreuzung. Jede Ampel wird durch eine ‘CardinalDirection’ identifiziert, somit haben wir für jede Himmelsrichtung eine Ampel.

Die Ampel Farben werden mit den Datentypen ‘sort Colour’ festgelegt (red, green, yellow). Jede Ampel startet mit der Farbe red und die Reihenfolge wechselt wie folgt: red -> green -> yellow -> red.

Definition der Daten Typen CardinalDirection und Colour

Für die Himmelsrichtung werden die Datentypen ‘CardinalDirection’ und ‘Axis’ angegeben. Eine ‘CardinalDirection’ kann die Werte ‘north’, ‘east’, ‘south’ und ‘west’ und die ‘Axis’ die Werte ‘nsAxis’ und ‘ewAxis’ annehmen.

```
sort
   CardinalDirection = struct north | east | south | west; % 4 directions
   Axis = struct nsAxis | ewAxis; % 2 axes
```

Das Mapping der Himmelsrichtung ist durch die Funktion ‘axis’ implementiert, dabei ist die ‘CardinalDirection’ der Übergabeparameter und als Wert wird eine ‘Axis’ zurückgegeben.

```
map
   axis : CardinalDirection -> Axis;
```

Hier wird die Semantik für die Datentypen ‘CardinalDirection’ und ‘Axis’ festgelegt. Somit wird bei der ‘CardinalDirection’ ‘north’ und ‘south’ die ‘Axis’ ‘nsAxis’ zurückgegeben und bei ‘east’ und ‘west’ die ‘Axis’ ‘ewAxis’.

```
eqn
   axis(north) = nsAxis;
   axis(south) = nsAxis;
   axis(east) = ewAxis;
```

```
    axis(west) = ewAxis;
```

Die Colour kann die Werte ‘red’, ‘yellow’ und ‘green’ annehmen.

```
sort
    Colour =  struct red | yellow | green;                % 3 colours
```

Für die Funktion ‘next’ wird eine ‘Colour’ als Parameter übergeben und eine ‘Colour’ zurückgegeben.

```
map
    next : Colour -> Colour;
```

Die Funktion ‘next’ ist so definiert, dass folgende Reihenfolge der Farben zurückgegeben wird: red -> green -> yellow -> red usw.

```
eqn
    next(red) = green;
    next(green) = yellow;
    next(yellow) = red;
```

Unteraufgabe: 2b.1

In der ersten Unteraufgabe haben wir die vier Ampeln parallel und unabhängig voneinander spezifiziert. Die Zustände (Farben) wechseln in einer Endlosschleife.

Definition TrafficLight Aktion und Prozesse

In der ersten Version der TrafficLight Aufgabe, wird nur eine Aktion benötigt. Das ist die ‘show-Aktion’, welche die Himmelsrichtung und die aktuelle Farbe der Ampel anzeigt.

```
act
    show : CardinalDirection # Colour;  % the given traffic light shows the given colour
```

Der obere ‘TrafficLight-Prozess’ mit den Parametern ‘CardinalDirection’ und ‘Axis’ initialisiert die Ampeln. Dabei wird der untere ‘TrafficLight-Prozess’ gestartet, hier wird die ‘CardinalDirection’ und die “Start-Farbe” als ‘Colour’ als Parameter übergeben. Der untere Prozess ruft erst ‘show’ auf, um den Zustand der aktuellen Ampel anzuzeigen. Sobald die ‘show-Aktion’ ausgeführt wird, ruft sich der ‘TrafficLight-Prozess’ selbst auf und somit kommt es zu einer Endlosschleife. Beim rekursiven Aufruf wird die Farbe weiter geschaltet.

```
proc
    TrafficLight(d : CardinalDirection, startAxis : Axis) =
        TrafficLight(d, red)
    ;

    TrafficLight(d : CardinalDirection, c : Colour) =
        % den aktuellen Zustand der Ampel anzeigen und zur naechsten Farbe wechseln
        show(d,c) . TrafficLight(d, next(c))
    ;
```

Definition Intersection

Die Prozesse der Ampeln werden gleichzeitig initialisiert und gestartet.

```
init
   TrafficLight(north, nsAxis) || TrafficLight(east, nsAxis) || TrafficLight(south, nsAxis) ||
TrafficLight(west, nsAxis)
;
```

Unteraufgabe: 2b.2

Für die zweite Aufgabe wird ein weiterer Prozess spezifiziert, der Monitor. Der Monitor wird parallel zu den ‘TrafficLight’ Prozessen ausgeführt und beobachtet die ‘show-Aktionen’. Dabei werden die aktuellen Ampel-Zustände überprüft, ob diese sicher sind. Sobald die Zustände als unsicher eingestuft werden, wird eine Fehlermeldung ausgegeben (intersectionUnsafe: Colour # Colour # Colour # Colour). Die Ampelschaltung ist dann sicher, wenn nur die Ampeln der gleichen Himmelsachsen auf grün oder gelb geschaltet werden. Als Beispiel, sobald die South-Ampel auf gelb gesetzt wird und die East-Ampel auf Grün, haben wir einen unsicheren Zustand.

Definition TrafficLight Prozesse

Dieser Prozess zeigt zuerst mit der ‘show-Aktion’ die “Start-Zustände” der Ampel an. Nach dieser Aktion wird der zweite ‘TrafficLight-Prozess’ aufgerufen.

```
proc
   TrafficLight(d : CardinalDirection, startAxis : Axis) =
      show(d, red) . TrafficLight(d, red) % Start Zustand/Farbe anzeigen
   ;
```

Der zweite Prozess zeigt mit der ‘show-Aktion’ die Ampel mit der nächsten Farbe an und ruft sich dann rekursiv mit der nächsten Farbe auf.

```
   TrafficLight(d : CardinalDirection, c : Colour) =
      % den aktuellen Zustand der Ampel anzeigen und TrafficLight mit naechster Farbe aufrufen
      show(d, next(c)) . TrafficLight(d, next(c))
   ;
```

Definition Monitor Aktionen und Prozess

Für den Monitor-Prozess haben wir drei weitere Aktionen spezifiziert. Die ‘recieve-Aktion’ mit der ‘CardinalDirection’ und ‘Colour’, die ‘intersectionUnsafe-Aktion’ mit vier Mal ‘Colour’ und die ‘colourSeen-Aktion’ mit der ‘CardinalDirection’ und die ‘Colour’.

```
act
   recieve : CardinalDirection # Colour;
   intersectionUnsafe : Colour # Colour # Colour # Colour;
   colourSeen : CardinalDirection # Colour;
```

Der Prozess des Monitors wird mit den aktuellen Zuständen der Ampeln parametrisiert. Sobald ein unsicherer Zustand auftritt, wird die Aktion ‘intersectionUnsafe’ aufgerufen und das System wird angehalten. Die Abfrage ‘safe‘ ist bei den Datentypen ‚Colour‘ spezifiziert.

```
map
   safe : Colour # Colour # Colour # Colour -> Bool;

var
   colourNorth : Colour;
   colourEast : Colour;
   colourSouth : Colour;
   colourWest : Colour;

eqn
   % wenn unsicherer Zustand, dann intersectionUnsafe
   safe(colourNorth, colourEast, colourSouth, colourWest) =
      ((!(colourNorth in {red}) || !(colourSouth in {red})) && (!(colourWest in {red}) || !(colourEast in
{red})));

proc
   Monitor(cNorth : Colour, cEast : Colour, cSouth : Colour, cWest : Colour) =
         % wenn unsicherer Zustand, dann intersectionUnsafe
         (safe(cNorth, cEast, cSouth, cWest)) -> intersectionUnsafe(cNorth, cEast, cSouth, cWest) <>
```

Falls der Zustand der Ampelschaltung sicher ist, werden alle Optionen der ‘Colour’ und ‘CardinalDirection’ durchlaufen. Die ‘recieve-Aktion’ wird gleichzeitig mit der ‘show-Aktion’ ausgeführt und besitzt somit die gleichen Werte wie die ‘show-Aktion’. Durch die Abfragen wird der ‘Monitor-Prozess’ mit der nächsten Farbe der aktuellen Ampel rekursiv aufgerufen.

```
         % wenn der Zustand sicher ist
         sum c : Colour, d : CardinalDirection. recieve(d, next(c)) . % Jede Zustands Moeglichkeit
durchgehen
            (d == north) -> Monitor(next(c), cEast, cSouth, cWest) <> % Zustand an den Monitor
uebergeben
            (d == east) -> Monitor(cNorth, next(c), cSouth, cWest) <>
            (d == south) -> Monitor(cNorth, cEast, next(c), cWest) <>
            (d == west) -> Monitor(cNorth, cEast, cSouth, next(c))
   ;
```

Definition Intersection

In dem Intersection-Prozess verwenden wir die Operatoren ‘hide’, ‘allow’ und ‘comm’. Der ‘hide-Operator’ wird verwendet, um internes Verhalten zu verstecken und leere ‘Multi-Aktionen’ werden durch tau ersetzt. ‘allow’ beschreibt die ‘Multi-Aktionen’ die ausgeführt werden dürfen.
In ‘comm’ werden Aktionen zu Multi-Aktionen verknüpft, sodass diese parallel und mit den gleichen Werten ausgeführt werden.

In diesem Fall wird der ‘hide-Operator’ nicht benötigt, da kein internes Verhalten versteckt werden muss. Mit dem ‘allow-Operator’ dürfen die ‘Multi-Aktionen’ ‘colourSeen’ und

‘intersectionUnsafe’ auftreten. In ‘comm’ werden ‘show’ und ‘recieve’ zu ‘colourSeen’ verknüpft.

Mit ‘TrafficLight(north, nsAxis) || TrafficLight(east, nsAxis) || TrafficLight(south, nsAxis) || TrafficLight(west, nsAxis) || Monitor(red,red,red,red)’ werden die Prozesse initialisiert und gleichzeitig gestartet.

```
   Intersection =
      %hide({

      %},
      allow({
         colourSeen,
         intersectionUnsafe
      },
      comm({
         show | recieve -> colourSeen % show und recieve gleichzeitig ausfuehren
      },
         TrafficLight(north, nsAxis) || TrafficLight(east, nsAxis) || TrafficLight(south, nsAxis) ||
TrafficLight(west, nsAxis) || Monitor(red, red, red, red)
      ))%)
   ;

init
   Intersection
;
```

Unteraufgabe: 2b.3

Bei der dritten Unteraufgabe wird durch den Monitor verhindert, dass unsichere Zustände auftreten. Unsichere Zustände sind z.B., wenn die North-Ampel und East-Ampel auf ‘green’ gesetzt werden.

Definition TrafficLight Prozesse

Bei dem ersten ‘TrafficLight-Prozess’ wird erst die Multi-Aktion ‘colourSeen’ aufgerufen, um den Ursprungszustand der Ampeln anzuzeigen. Erst danach wird der zweite ‘TrafficLight-Prozess’ mit den Ursprungszustand ‘red’ aufgerufen. Der zweite ‘TrafficLight-Prozess’ ist äquivalent zu dem Prozess in der Unteraufgabe 2b.2.

```
proc
   TrafficLight(d : CardinalDirection, startAxis : Axis) =
      colourSeen(d, red) . TrafficLight(d, red) % Start Zustand/Farbe anzeigen
   ;
```

Definition Monitor Aktionen und Prozess

Bei dem Prozess ‘Monitor’ haben sich die Parameter gegenüber der Unteraufgabe 2b.2 nicht verändert, jedoch die Abfrage, um zu überprüfen, ob der Zustand der Ampelschaltung sicher ist. Als Beispiel, sobald der Zustand der Ampel mit der Himmelsrichtung ‘north’

grün ist und die Ampeln mit der Himmelsrichtung ‘west’ und ‘east’ nicht grün oder gelb sind, ist der Zustand sicher. Die Abfrage ‘safe‘ ist bei den Datentypen ‚Colour‘ spezifiziert.

```
map
   safe : Colour # Colour # Colour # Colour -> Bool;

var
   colourNorth : Colour;
   colourEast : Colour;
   colourSouth : Colour;
   colourWest : Colour;

eqn
   % wenn unsicherer Zustand, dann intersectionUnsafe
   safe(colourNorth, colourEast, colourSouth, colourWest) =
      ((colourNorth in {green, yellow} || colourSouth in {green, yellow})
      && (colourWest in {green, yellow} || colourEast in {green, yellow}));

proc
   Monitor(cNorth : Colour, cEast : Colour, cSouth : Colour, cWest : Colour) =
      % wenn unsicherer Zustand, dann intersectionUnsafe
      (safe(cNorth, cEast, cSouth, cWest)) -> intersectionUnsafe(cNorth, cEast, cSouth, cWest)
```

Wenn der Zustand der Ampelschaltung sicher ist, werden auch hier alle Optionen von ‘Colour’ und ‘CardinalDirection’ durchlaufen. Die vier Abfragen sind immer nach dem gleichen Schema aufgebaut. Als Beispiel, wenn die ‘CardinalDirection’ gleich ‘north’ ist und die Ampeln links und rechts von der ‘north’ Ampel auf den Zustand ‘red’ geschaltet sind, kann fortgefahren werden. Somit kann die ‘recieve-Aktion’ mit der richtigen ‘CardinalDirection’ aufgerufen werden. Zum Schluss wird der ‘Monitor-Prozess’ mit dem neuen Zustand der Ampel rekursiv aufgerufen.

```
      <> % wenn Zustand sicher ist
      sum c : Colour, d : CardinalDirection. % Jede Zustands Moeglichkeit durchgehen
         (d == north && cEast == red && cWest == red) -> recieve(north, c) . Monitor(cNorth = c) <> %
Zustand an den Monitor uebergeben
         (d == east && cNorth == red && cSouth == red) -> recieve(east, c) . Monitor(cEast = c) <>
         (d == south && cEast == red && cWest == red) -> recieve(south, c) . Monitor(cSouth = c) <>
         (d == west && cNorth == red && cSouth == red) -> recieve(west, c) . Monitor(cWest = c)
   ;
```

Definition Intersection

Die Intersection der Unteraufgabe 2b.3 gleicht sich mit der Intersection der Unteraufgabe 2b.2.

Unteraufgabe: 2b.4

In der letzten Aufgabe geht es darum, ein verteiltes Verfahren für die Ampelschaltung zu entwickeln. Dabei werden die Ampeln als eigene Instanzen arbeiten. Die Ampeln werden mit einer Himmelsrichtung (sort CardinalDirection) spezifiziert und können miteinander kommunizieren, d.h. synchronisieren. Durch die Synchronisation werden unsichere Zustände verhindert.

Definition TrafficLight Aktionen und Prozesse

Die Prozesse ‘TrafficLight’ haben die Aktionen ‘show’ und ‘wasShown’ zur Verfügung. ‘wasShown’ dient zum Synchronisieren mit der gegenüberliegenden Ampel.

```
act
   show : CardinalDirection # Colour;  % the given traffic light shows the given colour
   wasShown : Axis;
```

Der erste ‘TrafficLight’ Prozess dient zum Initialisieren des zweiten ‘TrafficLight’ Prozesses und stellt mithilfe der gegebenen Himmelsrichtung und Startachse fest, ob der Prozess zu Beginn dran ist oder erst warten muss.

```
proc
   TrafficLight(d : CardinalDirection, startAxis : Axis) =
      % auf Startrichtung pruefen
      (axis(d) == startAxis) -> TrafficLight(d, red) <> waitDirection(d) . TrafficLight(d, red)
   ;
```

Der zweite ‘TrafficLight’ Prozess zeigt mithilfe von ‘show’ seine aktuelle Himmelsrichtung und Farbe an.
Anschließend wartet ‘wasShown’ auf die gegenüberliegende Himmelsrichtung. Wenn die aktuelle Farbe ‘red’ ist, wird erst die Aktion ‘nextDirection’ aufgerufen und dann auf die gegenüberliegende Himmelsrichtung gewartet ‘waitDirection’. Anschließend oder wenn die Farbe nicht ‘red’ war, wird der Prozess mit der nächsten Farbe wieder aufgerufen.

```
   TrafficLight(d : CardinalDirection, c : Colour) =
      % show(d, c): den aktuellen Zustand anzeigen
      % wasShown(axis(d)): auf die gegenueberliegende Ampel warten
      % (c == red): wenn Ampel rot
      % nextDirection(d): Richtungswechsel
      % waitDirection(d): auf Richtungswechsel warten
      show(d, c) . wasShown(axis(d)) . (c == red) ->
      nextDirection(next(d)) . waitDirection(d) . TrafficLight(d, next(c)) <> TrafficLight(d, next(c))
   ;
```

Die Multi-Aktion ‘axisWasShown’ kann erst ausgeführt werden, wenn die Achsen der Ampeln gleich sind. Es dient zur Synchronisation mit der gegenüberliegenden Ampel. Die Multi-Aktion ‘changedDirection’ synchronisiert die Aktionen ‘nextDirection’ und ‘waitDirection’, so müssen die Ampeln einer Achse so lange warten, bis die andere Achse weiter geschaltet hat.

```
act
   nextDirection : CardinalDirection;
   waitDirection : CardinalDirection;
   axisWasShown : Axis;
   changedDirection : CardinalDirection;
```

Definition Intersection

Die Aktionen ‘axisWasShown’ und ‘changedDirection’ werden versteckt da sie keine entscheidenden Punkte in der Ausführung darstellen.
Durch ‘allow’ dürfen die Aktionen und Multi-Aktionen ‘show’, ‘axisWasShown’ und ‘changedDirection’ auftreten.
Die Aktionen ‘waitDirection’ und ‘nextDirection’ werden zu der Multi-Aktion ‘changedDirection’ verknüpft sowie ‘wasShown’ zu der Multi-Aktion ‘axisWasShown’ wird.

```
   Intersection =
      hide({
         axisWasShown,
         changedDirection
      },
      allow({
         show,
         axisWasShown,
         changedDirection
      },
      comm({
         waitDirection | nextDirection -> changedDirection,
         wasShown | wasShown -> axisWasShown
      },
         TrafficLight(north, nsAxis) || TrafficLight(east, nsAxis) || TrafficLight(south, nsAxis) ||
TrafficLight(west, nsAxis)
      )))
   ;

init
   Intersection
;
```