Aufgabe 2:
Spezifikation in mCRL2

Erstellt von:

Ali Ben Brahem 7013448
Markus Luckau 7012601
Mosaab Helal 7012537
Simon Struck 7012333

Inhaltsverzeichnis

Aufgabe 2a - Sequenzielle Spezifikation

Problem

Es soll eine Verkaufsmaschine mit mCRL2 realisiert werden. Diese Maschine bietet folgende Produkte an: Tee für 10 Cent, Kaffee für 25 Cent, Kuchen für 60 Cent das Stück und einen Apfel für 80 Cent das Stück. Die Maschine soll folgende Münzen annehmen: 5 Cent, 10 Cent, 20 Cent, 50 Cent und 1 Euro. Sobald das Guthaben für ein Produkt ausreicht, soll es zum Verkauf angeboten werden. Sobald ein Produkt gekauft wurde soll es ausgegeben werden und das Guthaben dementsprechend angepasst werden. Sollte noch Guthaben vorhanden sein, wird dieses als Wechselgeld von groß nach klein ausgegeben. Der Automat kann nicht mehr als 200€ an münzen aufnehmen und muss danach geleert werden.

Lösung

Münzen
Das struct Coin wird erstellt und es werden ihm die Passenden werte zugeteilt

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

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

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

Produkte
Das struct Product wird erstellt und es werden ihm die passenden Werte zugeteilt

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

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

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

Aktionen

An diese Stelle im Code werden die einzelnen Aktionen, die die Verkaufsmaschine ausführen kann als Methoden deklariert. Die Aktion “accept” ist für das entgegennehmen von Münzen verantwortlich. Mit der Aktion “return” wird wechselgeld ausgegeben. Mit der Aktion “offer” wird bei einem ausreichenden Guthaben die entsprechenden Produkte gezeigt. Mit der Aktion “serve” giebt die Verkaufsmaschine einzelne Produkte aus. Die Aktion “returnChange” dient als zustand, wenn das Rückgeld nicht mit einer Münze ausgegeben werden kann.

```
% ------------------------------------------------------------------------
%
% Definition of the actions
%
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
```

Prozesse

<u>VendingMachine</u>

Die VendingMachine startet mit einem Credit von 0.
Von hier aus, können nur Münzen eingeworfen werden.
Nachdem eine Münze eingeworfen wurde, bietet die VendingMachine alle Produkte an, für die der Credit ausreicht und erlaubt es einem, den Credit in Münzen zurück zu erhalten.

```
%
% Definition of the processes
%
proc
    VendingMachine = VM(0);


    VM(credit : Int) =
        % if credit less than 200, accept more coins
        (credit < 200) -> sum c : Coin . accept(c) . VM(credit + value(c)) +
        % if there is enough credit for a product, offer it
        sum p : Product . (credit >= price(p)) -> offer(p) . serve(p) . VM(credit - price(p)) +
        % if there is any credit, offer to return it
        (credit > 0) -> returnChange(credit) . ReturnChange(credit)
    ;
```

ReturnChange

Der Prozess “ReturnChange” beschreibt den reibungslosen Ablauf der Ausgabe des Wechselgeldes. Das ziel dabei ist es das bestehende Restguthaben mit so wenig Münzen wie möglich auszuzahlen (große Münzen zuerst). Der Prozess besteht dabei aus einer IF-ELSE abfrage. Wenn das Guthaben größer als einen Euro wird ein Euro ausgegeben und der Prozess. Dies wird so lange wiederholt bis das Restguthaben unter einem Euro ist. Danach wird das gleiche mit der nächst kleineren Münze gemacht, bis das Restguthaben den Wert Null erreicht hat. Danach wird der Prozess VendingMachine gestartet.

```
ReturnChange(credit : Int) =
    % if remaining credit is greater than a euro, return a euro
    (credit >= value(Euro)) -> return(Euro) . ReturnChange(credit - value(Euro)) <>
    % else if remaining credit is greater than 50 cent, return a 50 cent coin
    (credit >= value(_50c)) -> return(_50c) . ReturnChange(credit - value(_50c)) <>
    % else if remaining credit is greater than 20 cent, return a 20 cent coin
    (credit >= value(_20c)) -> return(_20c) . ReturnChange(credit - value(_20c)) <>
    % else if remaining credit is greater than 10 cent, return a 10 cent coin
    (credit >= value(_10c)) -> return(_10c) . ReturnChange(credit - value(_10c)) <>
    % else if remaining credit is greater than 5 cent, return a 5 cent coin
    (credit >= value(_5c)) -> return(_5c) . ReturnChange(credit - value(_5c)) <>
    % else, return a new Vending machine with no credit
    VM(0)
;
```

Aufgabe 2b - Parallele Spezifikation

Problem Allgemein

Es soll eine Ampelsteuerung für eine Kreuzung mit mCRL2 realisiert werden. Die vier Ampeln besitzen die Farben rot gelb und grün. Die Abfolge der Farben beim Schalten der Ampeln lautet: rot, gelb, grün, rot. Jede Ampel wird einer Himmelsrichtung (norden, süden, osten, westen) zugewiesen.

Datentypen
Alle Versionen der Ampelsteuerung Benutzen die gleichen Datentypen. Das Struct “CardinalDirection” kann die Werte “north”, “east”, ”south” und ”west” annehmen. Dadurch werden die vier verschiedenen Standpunkte der einzelnen Ampeln beschrieben. Das Struct “Axis” kann die beiden Werte “nsAxis” und “ewAxis” annehmen, was für die beiden Richtungen, in die der Verkehr fließt, steht.
Zudem gibt es die Funktion “axis”, die eine “CardinalDirection” auf eine Axis mappt.

```
%
% Definition of data types
%

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

map
    axis: CardinalDirection -> Axis;

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

Das Struct “Colour” kann die Werte “red”, “yellow” und “green” annehmen. Diese stehen für die verschiedenen Farbzustände die die Ampelanlage annehmen kann. Zudem gibt es die Funktion “next”, die eine “Farbe” auf eine Colour mappt. Dadurch wird die in der Problembeschreibung vorgeben Farbreihenfolge eingehalten.

```
sort
    Colour =  struct red | yellow | green;

map
    next : Colour -> Colour;

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

Problem Version 1

Alle vier Ampeln sollen parallel voneinander in einer Endlosschleife alle Farbzustände durchlaufen.

Lösung

Die erste Version der Ampelsteuerung besitzt nur eine Aktion “show”, die die aktuelle Farbe der Ampel anzeigt. Die Ampel wird mit einer “CardinalDirection” versorgt und im Startzugang auf die Farbe Rot gesetzt. Wenn die “CardinalDirection” zusammen mit einer Farbe übergeben wird, wird die aktuelle farbe angezeigt und die Ampel mit der gleichen CardinalDirection und der nächsten Farbe neu gestartet.

```
%
% Definition of a TrafficLight
%

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

proc
    TrafficLight(d : CardinalDirection, startAxis : Axis) =
        % become a normal TrafficLight
        TrafficLight(d, red);

    TrafficLight(d : CardinalDirection, c : Colour) =
        % show light
        show(d, c) . TrafficLight(d, next(c));
```

Problem Version 2

Zusätzlich zu den Parallel laufenden Ampeln soll jetzt noch ein Prozess Monitor hinzugefügt werden, der eine Fehlermeldung ausgibt, sobald das Überqueren der Kreuzung nicht sicher ist und dadurch das Gesamtsystem anhält.

Lösung

Datentypen

Map

Die Map “Map” ist eine Map wo eine CardinalDirection einer Farbe zugewiesen wird.

```
%
% Definition of a Map
%

sort
    % sorts K symbolizes a "generic" Key sort and V symbolizes a "generic" Value sort,
    % that have to be instantiated later
    Map = CardinalDirection -> Colour;

map
    % _Map symbolizes the Map constructor,
    % the V element is used as default value to initialize undefined map entries
    _Map: Colour -> Map;
    start: Map;

var
    k: CardinalDirection;
    v: Colour;

eqn
    % initializes the map with the given default value (catch-all function)
    _Map(v)(k) = v;
```

Funktionen

isCrossingUnsafe

Die Funktion “isCrossingUnsafe” bekommt vier Farben mit übergeben und ermittelt mit hilfe eines Boolschen Ausdruckes, ob es nicht sicher ist sie Kreuzung zu überqueren. Es ist nicht sicher die Kreuzung zu überqueren, wenn die erste Farbe und die dritte Farbe beide nicht rot sind oder die zweite Farbe und die vierte Farbe nicht rot sind.

```
%
% Definition of the Function isCrossingUnsafe
%

map
    % returns a boolean that describes if the crossing is unsafe
    isCrossingUnsafe: Colour # Colour # Colour # Colour-> Bool;

var
    n, w, s, e : Colour;
eqn
    isCrossingUnsafe(n, w, s, e) = !((n == red && s == red) || (w == red && e == red));
```

Monitor

Der Monitor bietet alle Übergänge an, die möglich sind.
Die Kommunikation mit der “show”-Aktion der Traffic-Lights führt diese dann aus.
Somit kennt der Monitor immer den aktuellen Zustand aller TrafficLights.
Sollte die Funktion “isCrossingUnsafe” true zurückgeben, wird anstatt der Übergänge nur der Übergang “intersectionUnsafe” angeboten und danach läuft die Simulation in ein Deadlock.

```
%
% Definition of a Monitor
%

act
    seeColour: CardinalDirection # Colour;
    intersectionUnsafe: Colour # Colour # Colour # Colour;

proc
    Monitor(crossing: Map) =
        (isCrossingUnsafe(crossing(north), crossing(west), crossing(south), crossing(east))) ->
            intersectionUnsafe(crossing(north), crossing(east), crossing(south), crossing(west)) . delta
        <>
        (
            sum cd : CardinalDirection . (
                sum col : Colour . (
                    seeColour(cd, col) . Monitor(crossing[cd -> col])
                )
            )
        );
```

Zusätzlich zu den TrafficLights wird jetzt auch der Monitor in der Intersection gestartet.
Die einzigen zugelassenen Aktionen sind “colourSeen” und “intersectionUnsafe”, da sie die Kommunikation mit dem Monitor und das Deadlock darstellen.

```
%
% Definition of an Intersection with four traffic lights and a Monitor
%

act
    colourSeen : CardinalDirection # Colour;

proc
    Intersection =
        allow({
            colourSeen,
            intersectionUnsafe
        },
        comm({
            show | seeColour -> colourSeen
        },
             TrafficLight(north, nsAxis) ||
             TrafficLight(east, nsAxis) ||
             TrafficLight(south, nsAxis) ||
             TrafficLight(west, nsAxis) ||
             Monitor(_Map(red))
        ));
```

Problem Version 3

Der in Version 2 hinzugefügte Monitor soll nun das Auftreten unsicherer Kombinationen der Ampelanlage unterbinden.

Lösung

Datentypen

Map

Die Map “Map” ist eine Map wo eine CardinalDirection einer Farbe zugewiesen wird.

```
%
% Definition of a Map
%

sort
    % sorts K symbolizes a "generic" Key sort and V symbolizes a "generic" Value sort,
    % that have to be instantiated later
    Map = CardinalDirection -> Colour;

map
    % _Map symbolizes the Map constructor,
    % the V element is used as default value to initialize undefined map entries
    _Map: Colour -> Map;
    start: Map;

var
    k: CardinalDirection;
    v: Colour;

eqn
    % initializes the map with the given default value (catch-all function)
    _Map(v)(k) = v;
```

Funktionen

isCrossingUnsafe

Die Funktion “isCrossingUnsafe” bekommt vier Farben mit übergeben und ermittelt mit hilfe eines Boolschen Ausdruckes, ob es nicht sicher ist sie Kreuzung zu überqueren. Es ist nicht sicher die Kreuzung zu überqueren, wenn die erste Farbe und die dritte Farbe beide nicht rot sind oder die zweite Farbe und die vierte Farbe nicht rot sind.

```
%
% Definition of the Function isCrossingUnsafe
%

map
    % returns a boolean that describes if the crossing is unsafe
    isCrossingUnsafe: Colour # Colour # Colour # Colour-> Bool;

var
    n, w, s, e : Colour;
eqn
    isCrossingUnsafe(n, w, s, e) = !((n == red && s == red) || (w == red && e == red));
```

Monitor

Der Monitor hat eine kleine Änderung im Vergleich zu Version 2.
Hier bietet er jetzt nur noch Übergänge an, dessen Ausgang die Funktion “isCrossingSafe” true zurückgeben lässt.
Die Kommunikation mit den “show”-Aktionen der TrafficLights bleibt erhalten.
Somit wird gewährleistet, dass nur Zustände erreicht werden, die der Monitor als “safe” definiert.

```
%
% Definition of a Monitor
%

act
    seeColour: CardinalDirection # Colour;

proc
    Monitor(crossing: Map) =
        sum cd : CardinalDirection . (
            sum col : Colour . (
                isCrossingSafe(crossing[cd -> col]) ->
                    seeColour(cd, col) . Monitor(crossing[cd -> col])
            )
        )
    ;
```

Problem Version 4

In der Vierten Version soll die Ampelanlage als verteiltes System realisiert werden, bei dem es keine zentrale Steuereinheit gibt, sondern sich die vier Ampeln sich gegenseitig synchronisieren und so das Auftreten von unsicheren Kombinationen verhindern.

Lösung

Aktionen
Die vierte Version der Ampelanlage besitzt vier Aktionen.
Die Aktion “show” zeigt den aktuellen Farbzustand der Ampel an.
Die Aktion “becomeActive” wechselt eine Ampel in den aktiven Zustand, von dem sie auf grün wechseln darf.
Die Aktion “becomePassive” wird ausgeführt, wenn die aktive Achse durchgelaufen ist. Danach wird verhindert, dass sich die Farbzustände wechseln.
Die Aktion “sync” dient der Synchronisation zwischen den einzelnen Ampeln auf derselben Achse, da sonst eine Ampel von Rot -> Grün -> Gelb -> Rot schalten könnte, bevor die andere Ampel überhaupt von Rot -> Grün gewechselt hat.

```
%
% Definition of a TrafficLight
%

act
    show : CardinalDirection # Colour;  % the given traffic light shows the given colour
    becomeActive;                       % take control when other axis is done
    becomePassive;                      % hand over control when the axis is done
    sync: Axis;                         % synchronize the colours of the same axis
```

Prozess
TrafficLight
Jedes TrafficLight startet im oberen Prozess und wechselt dann auf den Unteren.
Wenn das TrafficLight auf der Startachse (NS) liegt, zeigt es Rot und wird danach Passiv.
Die TrafficLights auf der anderen Achse (EW) werden Aktiv, mit dem Passiv-werden der TrafficLights auf der Startachse.
Diese zeigen dann auch Rot und werden Passiv.
Das Passiv-werden, erlaubt den TrafficLights auf der Startachse mit dem Wechsel auf Grün zu beginnen.
Diese wechseln dann beide auf Grün und synchronisieren sich mit der Funktion “sync” auf ihre Achse, bevor sie auf die nächste Farbe wechseln.
Sobald sie auf Rot wechseln werden sie passiv, was der anderen Achse erlaubt aktiv zu werden.
Ab da wiederholt sich das Ganze dann ad infinitum.

```
proc
    TrafficLight(d : CardinalDirection, startAxis : Axis) =
        % starting axis can continue, other axes have to wait to take over control
        (axis(d) == startAxis) -> TrafficLight(d, red) <> becomeActive . TrafficLight(d, red);

    TrafficLight(d : CardinalDirection, c: Colour) =
         % show light
         show(d, c) . sync(axis(d)) . (
             % if it is red, handover control and get it back, otherwise synchronize
             (c == red) ->  becomePassive . sync(axis(d)) . becomeActive . TrafficLight(d, next(c))
             <>
             % finally continue with next colour
             TrafficLight(d, next(c))
         )
    ;
```

Intersection

Beim Prozess Intersection sind die Aktionen “colourCanged” und “swapActiveLight” versteckt, damit die “branching-bisim” das Programm mit der Referenzlösung vergleichen kann. Alle nicht nötigen Aktionen werden somit “versteckt”.
Die Aktionen “sync” und “sync” werden durch den “comm”-Operator zu “colourChaned” zusammengefasst, was eine Synchronisation der TrafficLights auf derselben Achse ermöglicht.
Die Aktionen “becomeActive” und “becomePassive” werden durch den “comm”-Operator zu “swapActiveLight” zusammengefasst, was eine Synchronisation zwischen den TrafficLights die aktiv werden und denen die passiv werden, damit die andere Achse auf Grün schalten kann.

```
%
% Definition of an Intersection with four traffic lights
%

act
    colourChanged: Axis;
    swapActiveLight;

proc
    Intersection =
        hide({
            colourChanged,
            swapActiveLight
        },
        allow({
            show,
            colourChanged,
            swapActiveLight
        },
        comm({
            sync | sync -> colourChanged,
            becomeActive | becomePassive -> swapActiveLight
        },
            TrafficLight(north, nsAxis) ||
            TrafficLight(east, nsAxis) ||
            TrafficLight(south, nsAxis) ||
            TrafficLight(west, nsAxis)
        )
    )
);
```