Eske Coordes
Gesine Krüger
Jan Behrends
Julian Hinxlage

Parallele und verteilte Systeme
Aufgabe 2

2a. Sequenzielle Spezifikationen

```
% VM.mcrl2
%
% A Simple Vending Machine.
%
% Copyright (c) 2019-2019 HS Emden-Leer
% All Rights Reserved.
%
% @version 1.00 - 01 Apr 2019 - GJV - initial version
%

%
-------------------------------------------------------------------------------------------------------------------
-------
%
% Definition of the coins
%
sort
        Coin = struct _5c | _10c | _20c | _50c | Euro;      %5 different coins

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

eqn             %Zuweisung der Coins mit Integer-Werten
        value(_5c) = 5;
        value(_10c) = 10;
        value(_20c) = 20;
        value(_50c) = 50;
        value(Euro) = 100;
        next(_5c) = 10;        %Zuweisung welche Münze größer als die vorherige ist
        next(_10c) = 20;
        next(_20c) = 50;
        next(_50c) = 100;
        next(Euro) = 1000000; % should actualy be infinite

%
-------------------------------------------------------------------------------------------------------------------
-------
%
% Definition of the products
%
sort
        Product = struct tea | coffee | cake | apple;% 4 different products

map
```

Eske Coordes
Gesine Krüger
Jan Behrends
Julian Hinxlage

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

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

%
-------------------------------------------------------------------------------------------------------------------
-------
%
% 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

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

        VM(credit : Int) =
                % ist credit kleiner als 200, kann ein weiterer Coin akzeptiert werden und der Wert wird credit
hinzugefuegt
                (credit < 200) -> sum c : Coin.accept(c).VM(credit + value(c))
                % es kann auch ein Produkt ausgewählt werden, das bestellt und serviert wird und dessen Wert
anschließend von credit abgezogen wird
                + sum p: Product.(
                        (credit >= price(p)) -> offer(p).serve(p).VM(credit - price(p))
                )
                % ist credit größer as 0, kann die Aktion returnChange aufgerufen werden, die den Prozess
ReturnChange startet
                + (credit > 0) -> returnChange(credit).ReturnChange(credit);

        ReturnChange(credit : Int) =
                % für jede Art Coin (c) wird verglichen, ob credit größer gleich c ist und kleiner als das nächst
größere c.
                % falls ja, wird der Wert von c von credit abgezogen. Ist credit größer als 0 wird dieser Vorgang
wiederholt.
                % ist credit gleich 0 wird die VM mit 0 wieder von vorn gestartet
                sum c : Coin.((credit >= value(c) && credit < next(c)) -> (return(c).(
                        ((credit - value(c)) > 0) -> ReturnChange(credit - value(c)) <> VM(credit - value(c))
                )));
```

Eske Coordes
Gesine Krüger
Jan Behrends
Julian Hinxlage

```
%
-------------------------------------------------------------------------------------------------------------------
-------

init
        VendingMachine;
```

Beschreibung:
Zu Beginn der Spezifikation werden die Coins definiert und ihnen Integer-Werte zugewiesen. Welche Werte genau und welche Werte die nächstgrößeren sind, wird im Abschnitt *eqn* festgelegt. Da die Werte nicht größer als 1 Euro sein sollen, wird ein entsprechend großer Wert für next(Euro) gewählt.
Anschließend werden die Produkte definiert und mit verschiedenen Integer-Werten belegt.
Als nächstes folgt die Definition der Aktionen. Es gibt fünf verschiedene: accept, return, offer, serve und returnChange. accept akzeptiert einen Coin und return einen Coin zurück. Mit offer wird ein Produkt bestellt und mit serve serviert. returnChange ist die Anfrage das Wechselgeld auszugeben.
Nach der Definition der Aktionen folgt die Definition der Prozesse. Der Hauptprozess VM(credit : Int) wird mit credit aufgerufen. Ist credit kleiner als 200, kann die Aktion Coin.accept(c) aufgerufen werden, mit der eine Münze mit dem Wert c von der Maschine akzeptiert wird. Anschließend wird VM mit dem auf credit addierten Wert c erneut aufgerufen. Soll keine neue Münze eingeworfen werden, kann ein Produkt gekauft werden, sobald credit größer als der Preis des Produktes ist. Danach wird mit der Aktion offer(p) das übergebene Produkt gekauft und mit serve(p) das entsprechende Produkt serviert. Mit VM(credit - price(p)) wird der Preis des Produktes von credit abgezogen und VM erneut aufgerufen.
Ist credit größer als 0, kann das Wechselgeld ausgegeben werden. Dazu wird die Aktion returnChange ausgeführt, damit ReturnChange aufgerufen werden kann. ReturnChange wird credit übergeben und die Münzen der Größe nach absteigend ausgegeben. Der Wert für c startet bei 5 und solange credit größer gleich c und kleiner als das nächst größere c ist, wird c ausgegeben. Der Wert von c wird von credit abgezogen und solange credit größer ist, kann entweder ReturnChange mit dem aktualisierten Wert für credit aufgerufen werden oder VM mit 0 neu gestartet werden.

Aufgabe 2b. Parallele Spezifikation

2b_1

```
sort
        CardinalDirection = struct north | east | south | west; %Himmelsrichtungen der Ampeln
        Colour = struct red | yellow | green; %Ampelfarben

map
        next: Colour -> Colour;

eqn
        next(red) = green;     %Gibt jeweils die folgende Ampelfarbe zurück
        next(green) = yellow;
        next(yellow) = red;
```

Eske Coordes
Gesine Krüger
Jan Behrends
Julian Hinxlage

```
act
	show: CardinalDirection # Colour;	%Action zur Anzeige der Ampelfarbe der Ampeln

proc
	TrafficLight(c : Colour, d : CardinalDirection) =
		show(d,c).TrafficLight(next(c),d);	%Zeigt die derzeitigen Ampel/Ampelfarben-Kombination an und
wechselt

init
	TrafficLight(red,north) ||	%Initialisierung mit den 4 Ampeln, auf rot stehend
	TrafficLight(red,east ) ||
	TrafficLight(red,south) ||
	TrafficLight(red,west ) ;
```

Beschreibung:
Aufgabe war es zunächst eine Spezifikation zu entwickeln, bei der 4 Ampeln parallel und unabhängig voneinander über den Prozess TrafficLight die drei möglichen Zustände (red, yellow, green) in Endlosschleife durchlaufen. Für die Ampeln wurden 2 Datenstrukturen CardinalDirection (Himmelsrichtung) und Colour (Farbe) spezifiziert. Über die next(Colour)-Gleichung wird die jeweils folgende Ampelphase zurückgegeben, um einen Zyklus für die Ampelphasen zu erhalten (red,yellow,green,red,yellow….) . Die show-Action zeigt die Himmelsrichtung sowie die derzeitige Ampelfarbe an. Mit dem Trafficlight-Process werden die derzeitigen Ampeln mit ihren jeweiligen Ampelfarben angezeigt und anschließend folgt ein rekursiver Aufruf von TrafficLights mit der folgenden Ampelfarbe (mit hilfe der next-Gleichung).

2b_2

```
sort
	CardinalDirection = struct north | east | south | west;	%Himmelsrichtungen der Ampeln
	Colour = struct red | yellow | green | none; %Ampelfarben

map
	next: Colour -> Colour;
	check: Colour # Colour # Colour # Colour -> Bool;
	go: Colour -> Bool;
	colourOnly: Colour -> Colour;

var
	c1,c2,c3,c4 : Colour;

eqn
	next(none) = red;	%Gibt jeweils die folgende Ampelfarbe zurück
	next(red) = green;
	next(green) = yellow;
	next(yellow) = red;

	colourOnly(none) = red;	%Gibt aktuelle Ampelfarbe zurück, außer bei none
	colourOnly(red) = red;
	colourOnly(yellow) = yellow;
	colourOnly(green) = green;
```

Eske Coordes
Gesine Krüger
Jan Behrends
Julian Hinxlage

```
	go(c1) = (c1 == green || c1 == yellow); %grün oder gelb?

	check(c1,c2,c3,c4) =
		if(
			((go(c1) || go(c3)) && (go(c2) || go(c4)))   %Ampelkonstellation zulässig?
			,false
			,true
		);

act
	show: CardinalDirection # Colour;
	crossingUnsafe: Colour # Colour # Colour # Colour;
	mon: CardinalDirection # Colour;

proc
	TrafficLight(d : CardinalDirection,c : Colour) =
		show(d,next(c)).TrafficLight(d,next(c));

	Monitor(c1,c2,c3,c4 : Colour) =
		!check(c1,c2,c3,c4)
		-> crossingUnsafe(colourOnly(c1),colourOnly(c2),colourOnly(c3),colourOnly(c4)).delta      %Wenn
nicht zulässig Deadlock
		<> (    %Ansonsten Auswahl aus diesen Aktionen
				mon(north, next(c1)).Monitor(c1=next(c1))+
				mon(east , next(c2)).Monitor(c2=next(c2))+
				mon(south, next(c3)).Monitor(c3=next(c3))+
				mon(west , next(c4)).Monitor(c4=next(c4))
			)
		;

act
	colourSeen: CardinalDirection # Colour;

init
	allow(
		{
			colourSeen,           %Erlaubte Actions
			crossingUnsafe
		},
	comm(
		{
			show|mon -> colourSeen
		},
		TrafficLight(west ,none) ||
		TrafficLight(south,none) ||
		TrafficLight(east ,none) ||
		TrafficLight(north,none) ||
		Monitor(none,none,none,none)
	));
```

Eske Coordes
Gesine Krüger
Jan Behrends
Julian Hinxlage

Beschreibung:
Der TrafficLights-Process macht dasselbe wie bei 2b_1. Der Monitor-Process überprüft, ob die derzeitigen Ampeleinstellungen zulässig sind. Wenn das nicht der Fall ist, wird ein Deadlock ausgelöst und mit crossingUnsafe wird die derzeitige (unzulässige) Ampeleinstellung angezeigt. Wenn die Ampeleinstellung zulässig ist, werden die neuen Aktionen zur Änderung der Ampeleinstellungen angezeigt.
Die Überprüfung der Ampeleinstellung erfolgt über die check-Gleichung. Ist bei den Ampelpaaren Norden/Süden und Westen/Osten mindestens jeweils eine Ampel grün oder gelb, liegt eine unzulässige Ampeleinstellung vor. In diesem Fall gibt check "false" zurück, ansonsten true. Damit die Ampeln in den Prozessen TrafficLight und Monitor synchronisiert sind, werden in der Initialisierung nur jene Aktionen erlaubt, bei denen show und mon identisch/synchronisiert sind (colourSeen).

2b_3

```
sort
	CardinalDirection = struct north | east | south | west;
	Colour = struct red | yellow | green | none;

map
	next: Colour -> Colour;
	check: Colour # Colour # Colour # Colour -> Bool;
	go: Colour -> Bool;

var
	c1,c2,c3,c4 : Colour;

eqn
	next(none) = red;	%Gibt jeweils die folgende Ampelfarbe zurück
	next(red) = green;
	next(green) = yellow;
	next(yellow) = red;

	go(c1) = (c1 == green || c1 == yellow);

	check(c1,c2,c3,c4) =
		if(
			((go(c1) || go(c3)) && (go(c2) || go(c4))) %Ampelkonstellation zulässig?
			,false
			,true
		);

act
	show: CardinalDirection # Colour;
	mon: CardinalDirection # Colour;

proc
	TrafficLight(d : CardinalDirection,c : Colour) =
		show(d,next(c)).TrafficLight(d,next(c));

	%prüfen der Sicherheit von Folgekombinationen
	Monitor(c1,c2,c3,c4 : Colour) =
		check(next(c1),c2,c3,c4) -> mon(north, next(c1)).Monitor(c1=next(c1))+
```

Eske Coordes
Gesine Krüger
Jan Behrends
Julian Hinxlage

```
                check(c1,next(c2),c3,c4) -> mon(east , next(c2)).Monitor(c2=next(c2))+
                check(c1,c2,next(c3),c4) -> mon(south, next(c3)).Monitor(c3=next(c3))+
                check(c1,c2,c3,next(c4)) -> mon(west , next(c4)).Monitor(c4=next(c4));

act
        colourSeen: CardinalDirection # Colour;

init
        allow(
                {
                        colourSeen
                },
        comm(
                {       %Synchronisation
                        show|mon -> colourSeen
                },

                TrafficLight(west ,none) ||
                TrafficLight(south,none) ||
                TrafficLight(east ,none) ||
                TrafficLight(north,none) ||
                Monitor(none,none,none,none)
        ));
```

Beschreibung:
In 2b_3 sollte die Spezifikation so abgeändert werden, dass das Auftreten von unsicheren Kombinationen durch den Monitor verhindert werden werden soll, wodurch die Funktion crossingUnsafe überflüssig wird. Der wesentliche Unterschied liegt somit in dem Monitor-Prozess. Um ein Deadlock zu verhindern, wird statt wie in 2b_2 nicht der aktuelle Zustand, sondern alle möglichen Folgezustände mit der check Funktion, auf eine sichere Kombination geprüft, und nur die Schaltvorgänge ermöglicht, für die ein true zurückgeliefert wird.
Da der Prozess mit einer sicheren Kombination Initialisiert wird und nur sichere Folgezustände ermöglicht werden, kann kein Deadlock durch unsichere Kombinationen nicht mehr entstehen.

Eske Coordes
Gesine Krüger
Jan Behrends
Julian Hinxlage

2b_4

```
sort
        CardinalDirection = struct north | east | south | west;
        Colour = struct red | yellow | green | none;

map
        next: Colour -> Colour;

eqn
    % Reihenfolge der Farben definieren
        next(none) = red;
        next(red) = green;
        next(green) = yellow;
        next(yellow) = red;

act
        show: CardinalDirection # Colour;
    % zur Synchronisierung der Aktiven Richtung
        changeActive;
        changePhase;
    % zur Synchronisierung der aktellen Farbe der Aktiven Richtung
        changeActiveAll;
        changePhaseBoth;

proc
        TrafficLight(d: CardinalDirection) =
        (d == north || d == south)
                 %TrafficLight wird mit none aufgerufen, falls north oder south dran sind
                -> TrafficLight(d, none)
                 %sind east oder west dran wird erst changeActive aufgerufen und dann TrafficLight
                <> changeActive.TrafficLight(d, none);


        TrafficLight(d : CardinalDirection,c : Colour) =
         % aktuelle Farbe anzeigen
        show(d, next(c)).(
                (next(c) == red)
                        %ist die nächste Farbe rot, wird zweimal changeActive aufgerufen, um die Ampeln zu
synchronisieren
                -> changeActive.changeActive.TrafficLight(d, next(c))
                        %ist die nächste Farbe nicht rot, wird nur die Phase geändert
                <> changePhase.TrafficLight(d, next(c))
        );


        Crossing =
        hide(
                {    % Synchronisations Actionen verbergen
                changeActiveAll,
                changePhaseBoth
                },
```

Eske Coordes
Gesine Krüger
Jan Behrends
Julian Hinxlage

```
	allow(
		{
			% keine Einzelnen Synchronisations Actionen erlauben
		show,
		changeActiveAll,
		changePhaseBoth
		},
	comm(
		{   %zur Synchronisation der Ampeln
			% alle Ampeln müssen gleichzeitig die Aktive Richtung ändern
		changeActive|changeActive|changeActive|changeActive -> changeActiveAll,
			% beide Ampeln der Aktiven Richtung müssen die Aktuelle Farbe Synchronisieren
		changePhase|changePhase -> changePhaseBoth
		},
		 %Ein Prozess für jede Himmelsrichtung
		TrafficLight(west) ||
		TrafficLight(south)||
		TrafficLight(east) ||
		TrafficLight(north)
	)));

init
	Crossing;
```

Beschreibung:
Für 2b_4 sollte die Spezifikation so abgeändert werden, dass ein Verteiltes Verfahren mit den Ampeln als eigenständig synchronisierte Prozesse entsteht. Der Monitor als zentrale Instanz und die Aktion mon fallen somit weg. Zusätzlich soll in Nord-Süd Richtung mit der ersten Grünphase begonnen werden. Um die Ampeln und deren Ampelphasen miteinander zu synchronisieren wurden die Spezifikation um die Aktionen changeActive, changePhase, changeActiveAll und changePhaseBoth erweitert. Die Aktion changeActive ändert, ob eine Ampel im aktiven(Schaltvorgänge) oder inaktiven (keine Schaltvorgänge) Zustand ist. Die Aktion changePhase übernimmt die eigentlichen Schaltvorgänge der Ampelphasen aktiver Himmelsrichtungen. changeActiveAll synchronisiert den wechsel der Ampeln von aktiv zu inaktiv, sodass alle ampeln gleichzeitig die Aktion changeActive ausführen. Das synchrone Schalten durch die Ampelphasen, der jeweils beiden aktiven Ampeln wird über changePhaseBoth synchronisiert. Der Prozess TrafficLight prüft zunächst, ob die Ampel aktiv ist. Ist die Ampel inaktiv, wird eine changeActive dieser Ampel ausgeführt. Da changeActive durch changeActiveAll synchronisiert wird wartet die Ampel solange, bis auch alle anderen Ampeln die Aktion changeActive ausführen. Ist die Ampel jedoch im aktiven Zustand kann von Rot zu Grün und von Grün zu gelb geschaltet werden (changePhase). Beim Schalten von Gelb zu Rot wird zusätzlich die Aktion changeActive ausgeführt um die Ampel in den inaktiven Zustand zu versetzen.