

BACHELOR INFORMATICA

UvA  UNIVERSITEIT VAN AMSTERDAM

# Software Engineering with PSF and Go

Erik van der Schaaf

June 8, 2016

**Supervisor(s):** Bob Diertens (UvA)

**Signed:** Bob Diertens (UvA)



## **Abstract**

In this thesis I designed an application using the software engineering with process algebra method. I use PSF for a high level of abstraction and Go for a low level of abstraction. The system I implemented is based on the BHS located at Amsterdam Airport Schiphol. This system consists of multiple components working together on a concurrent level.

I specified the system in PSF, and refined this specification into an implementation in Go. In the refinement from PSF to Go I found several patterns.



# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Outline . . . . .	6
<b>2</b>	<b>Process Specification Formalism</b>	<b>7</b>
2.1	Background . . . . .	7
2.2	Used functionalities . . . . .	7
2.2.1	Modules . . . . .	8
2.2.2	Sorts and functions . . . . .	8
2.2.3	Variables and equations . . . . .	9
2.2.4	Processes . . . . .	9
2.2.5	Atoms . . . . .	9
2.2.6	Sets . . . . .	9
2.2.7	Communication and encapsulation . . . . .	10
2.2.8	Definitions . . . . .	10
<b>3</b>	<b>Go</b>	<b>13</b>
3.1	Background . . . . .	13
3.2	Used functionalities . . . . .	13
3.2.1	Functions and imports . . . . .	14
3.2.2	Constants, structures and variables . . . . .	14
3.2.3	If...Else statements and for loops . . . . .	15
3.2.4	Concurrency and channels . . . . .	16
<b>4</b>	<b>Baggage handling system</b>	<b>19</b>
4.1	Schiphol . . . . .	19
4.2	My baggage handling system . . . . .	19
4.2.1	Check-in . . . . .	20
4.2.2	Screening . . . . .	20
4.2.3	Transport and sorting . . . . .	20
4.2.4	Transfer baggage . . . . .	21
<b>5</b>	<b>Implementation</b>	<b>23</b>
5.1	The process . . . . .	23
5.2	PSF . . . . .	23

5.2.1	Step 1: Transporting over a single conveyor belt . . . . .	25
5.2.2	Step 2: Sorting machines sending in a random direction .	27
5.2.3	Step 3: Sorting machines sending in a specified direction .	29
5.3	Go . . . . .	31
5.3.1	Step 1: Transporting over a single conveyor belt . . . . .	33
5.3.2	Step 2: Sorting machines sending in a random direction .	34
5.3.3	Step 3: Sorting machines sending in a specified direction .	36
<b>6</b>	<b>Results</b>	<b>39</b>
6.1	Processes . . . . .	39
6.2	Communication . . . . .	40
6.3	Equations . . . . .	40
<b>7</b>	<b>Conclusion</b>	<b>43</b>
	<b>Bibliography</b>	<b>43</b>

# Chapter 1

## Introduction

I engineered and developed a program using the software engineering with process algebra (SE-PA) method [1]. This method uses multiple levels of abstraction. I started on a high level of abstraction and worked my way to the lower levels of abstraction. With the use of the SE-PA method I was able to deal with the complexity of the problem and the solution. On these high levels of abstraction I described the behaviour of the system using process algebra. I used the Process Specification Formalism (PSF) [2] for this. I sought a way to implement these different specifications on multiple levels of abstraction to the programming language Go.

PSF comes with an important Toolkit. One of its tools is a simulator that can be coupled with an animation of the specification. [3] I used PSF because it allows me to start testing in an early stage of the design process using this simulator. The opportunity to test early in the design process prevents from errors getting into a lower level of abstraction.

The reason why I chose Go as target language for the specification in the lower abstraction levels, is the ease of implementing concurrent functions. This relatively new programming language uses channels, making it possible for concurrent functions to communicate. All these functionalities I used are built-in with Go and thus easy to use.

As subject for my program I used the baggage handling system (BHS) of Amsterdam Airport Schiphol [4]. There are a lot of different components working together on a concurrent level, with a lot of communication between these components. Most of the work is automated and the system is suitable to split into different levels of abstraction.

My research question is to develop an application with the use of the software engineering with process algebra method for a baggage handling system with Go as target programming language.

## 1.1 Outline

First I will explain some more about PSF in chapter 2. In chapter 3 I tell more about the programming language Go. Next, in chapter 4, I will explain the different components of the BHS I implemented. Chapter 5 presents my implementation of the application. In chapter 6 I will show the results I found during my research. Chapter 7 will conclude this thesis.



## Chapter 2

# Process Specification Formalism

In this chapter I will give a short description about Process Specification Formalism (PSF). I will also explain some functions I used from PSF for my program.

### 2.1 Background

In 1985 Sjouke Mauw started his research in the field of algebraic techniques for software specification. The goal was to create a tool that could improve software development using process algebra by aiding in specification, simulation, verification and implementation, or even automate it. As input language for such tools Sjouke Mauw designed PSF. His doctoral thesis covers this research [2].

PSF is still used at the University of Amsterdam nowadays. PSF supports Algebra of Communication Processes (ACP) and the part of PSF that deals with the description of data is based on Algebraic Specification Formalism (ASF). The main use for PSF is in communication protocols, but it can also be used in the specification of different systems. Examples and more information can be found on the website of PSF [3].

### 2.2 Used functionalities

I will not cover all the functionalities of PSF, but I will explain what I have used. For a description covering all of PSF I refer to the PhD thesis of Sjouke Mauw [2].

## 2.2.1 Modules

PSF has two different types of modules, the data module and the process module. A module consists of a predefined order of sections. Words in italics are identifiers that should be filled in. I use  $\dots$  to represent a list of elements. The different sections are explained in the chapters that follow.

```
1 data module Module
2 begin
3   sorts
4   ...
5   functions
6   ...
7   variables
8   ...
9   equations
10  ...
11 end Module
12
13 process module Module
14 begin
15   atoms
16   ...
17   processes
18   ...
19   sets
20   ...
21   communications
22   ...
23   variables
24   ...
25   definitions
26   ...
27 end Module
```

## 2.2.2 Sorts and functions

The sorts section consists of a comma separated list, like the one below.

```
1 sorts
2   S,
3   DATA
```

Functions work with these sorts declared in a data module. It is possible to make function calls with arguments, they should be separated by a hashtag (#). The result of the function is indicated with an arrow (->).

```
1 functions
2   f : -> S
3   f : DATA -> S
4   f : S # DATA -> S
```

### 2.2.3 Variables and equations

The next section is the variable section. This section varies whether it is used in the data module or the process module. Inside the data module the variable section lists the variables used in the equations. When used in the process module the variable section lists the variables used in the process definitions.

```
1 variables
2 x : -> S
```

Equations can give a definition of a function. PSF is a language with a term rewrite system. Equations apply this rule and so the left hand side is rewritten to the right hand side. It is also possible to use variables defined in the variable section. Below I have given two examples for defining an and function. The tags in front of the equation are for documentation purposes only and it does not matter what you fill in.

```
1 equations
2 [and1] and(false, false) = false
3 [and2] and(false, true) = false
4 [and3] and(true, false) = false
5 [and4] and(true, true) = true
6
7 [and1] and(false, x) = false
8 [and2] and(true, x) = x
```

### 2.2.4 Processes

In the process section you can declare the processes. Processes can have arguments separated by a hashtag.

```
1 processes
2 p
3 p : S
4 p : DATA # S
```

### 2.2.5 Atoms

Atoms are actions that processes can execute. Atoms are declared the same way as processes, arguments are separated by hashtags.

```
1 atoms
2 a
3 a : S
4 a : DATA # S
```

### 2.2.6 Sets

Sets contain sub-sections, the sets indicate the type of the set declared in his sub-section. There are various ways to fill a set as seen below.

```

1 sets
2   of atoms
3     H = set-expression
4   of S
5     D = set-expression
6     E = set-expression
7
8 Set-expressions:
9   sort S
10    all elements of the sort S
11   set S
12    all elements of the set S
13   enumeration { $e_1, e_2, \dots, e_n$ }
14    placeholders can be used
15    H = {a(x), b(y) | x in S, y in DATA}
16   union S + T
17   intersection S · T
18   difference S \ T

```

## 2.2.7 Communication and encapsulation

The communication section contains two communication partners separated by a '|' and the resulting action of that communication. This form of communication is used so that processes have to wait for both communications to be done, before they can perform the same action again.

```

1 communications
2   a | b = c
3   a(x) | b(x) = c(x) for x in S
4   a(x) | b(y) = c(x, y) for x in S, y in S

```

Encapsulation is used to limit the actions for process expressions. In the example below only the process can only execute actions from the process expression  $x$  that are not an element of set H. The definitions section is displayed in the next section.

```

1 definitions
2   P = encaps(H, x)

```

## 2.2.8 Definitions

The definitions section consists of process definitions, on the left hand side it states the process and on the right side its definition. The processes can have variables from the variables section as arguments. The list of process expressions is long, so I only list the ones that I have used.

```

1 definitions
2   P = process-expression
3   P(x) = process-expression
4   P(f(b), b(y)) = process-expression
5

```

6 **Process-expressions** (*some of them*):

7 **atomic action** a

8     Executes atomic action a

9 **deadlock** delta

10     Deadlocks can not be executed

11 **process** P

12     The process P is replaced with the definition of that

13     process in the definition section.

14 **sequential composition** x.y

15     First process expression x is executed, upon termination

16     process expression y is executed.

17 **alternative composition** x+y

18     One of the process expressions x and y is executed. The choice

19     is random, but choosing a deadlock is forbidden.

20 **parallel composition** x||y

21     process expressions x and y are executed parrallel, this allows

22     communications to exist.

23 **generalized alternative composition** sum(v in S,x)

24     The process expression x in which v is replaced by the value,

25     is executed as an alternative composition, for every

26     value of v in the sort or set S.

27 **generalized parallel composition** merge(v in S,x)

28     This is almost the samen as a generalized alternative

29     composition, but the process expression x is executed

30     as an parrallel composition instead of

31     an alternative composition.

32 **conditional expression** [t=u] -> x

33     If the terms t and u are equal, process expression x is

34     executed. If the terms are not equal the process

35     expression contains deadlock.



# Chapter 3

## Go

In this chapter I will explain who is developing Go language and why they are developing Go in the first place. I will also explain some functions I used from the Go language.

### 3.1 Background

Go or golang is a programming language created by Google engineers. Go is an open source project that became public on November 10, 2009 [5]. The first stable release of Go 1 was released on March 28, 2012. The current Go 1.6 is released in February of 2016 [6]. The motivation for creating this new programming language was the waiting time for a large Google server to compile. The most important goal for Go from the start was to be able to build Go code using only the source itself, no makefiles or a modern replacement for makefile is needed [7]. "*Go is efficient, scalable, and productive.*" is what Rob Pike said in his keynote talk at the SPLASH 2012 conference in Tucson, Arizona, on October 25, 2012 [8]. Rob Pike is one of the creators of Go.

### 3.2 Used functionalities

Go has a lot of functionalities thanks to the project being open source. Because there are so many, I will only go over the functionalities I have used for my program. There is a documentation on the website of Go for all the functionalities [9]. The book written by Mark Summerfield "Programming in Go" also covers a lot about the programming language Go [10].

It is possible to use comments in the code. Everything after `//` is seen as a comment and thus ignored.

### 3.2.1 Functions and imports

If you execute a Go program it will always start with the main function. So every program should have a main function and package main. You could also create more additional functions beside the required main function. There is also room for importing more packages for more functionality. If you import new packages, you will be able to use their functionalities on top of the built-in functionalities.

```
1 // Every Go program needs a package main
2 package main
3
4 // Importing multiple packages
5 import (
6     "packages 1"
7     "packages 2"
8     "path/filepath 1"
9     "path/filepath 2"
10 )
11
12 // The main function
13 func main() {
14
15 }
16
17 // A standard function layout
18 func functionName(arguments) {
19
20 }
```

### 3.2.2 Constants, structures and variables

Another standard functionality are constants and variables. It is possible to declare them at a global (package) or a function level. You need to state the type of the variable or constant. If you do not declare a variable but you only initialize the variable, then Go will automatically declare a variable with the type you are trying to assign to the variable. Constants need to be declared and initialized the first time you use them, because constants can not change value. Inside functions you can initialize variables with a second method, the short assignment statement ":=". Declaration is skipped and automatically done by Go. The second method does not work with constants or outside functions.

```
1 // Simple variable
2 var varName int
3 var varName2 string
4
5 // Simple constant
6 const constName int = 1
7 const constName2 string = "constant 2"
8
9 // Same can be done in a function
10 func functionName(arguments) {
11     var varName3 int
```



```

12  var varName4 string
13
14  varName5 := 3
15  varName6 := "variable 6"
16
17  const constName3 int = 3
18  const constName4 string = "constant 4"
19 }

```

Object oriented programming is a well known concept. Go makes object oriented programming possible with structures. These structures are collections of fields. You can give these structures your own type. Declaring your own types opens up a lot of new possibilities, because you are not bound to the standard variable types. It is also possible to make new functions working with these structures. These functions use a structure to operate and can only be executed with the use of a structure.

```

1 // Simple structure
2 type structType struct {
3     variableName int
4     variableName2 string
5 }
6
7 // A function working with the structure
8 func (structName *structType) functionName() {
9     // Here you could do something like print struct
10    fmt.Println("Int: ", structName.variableName)
11    fmt.Println("String: ", structName.variableName2)
12 }
13
14 // This is how you execute the function
15 test := structType{
16     variableName: 3,
17     variableName2: "Test",
18 }
19 test.functionName()

```

### 3.2.3 If...Else statements and for loops

The if and else statements are pretty standard. The expression need not be surrounded by parentheses but the braces are required.

```

1 if x == 0 {
2     // x is zero
3 } else if x > 0 {
4     // x is greater than zero
5 } else {
6     // x < 0
7 }

```

The for loop is the only looping construct Go has. The for loop has three components and a body. The components are separated by semicolons. The first component is the init statement, this component is executed before the iteration

and usually contains a short variable declaration. The second component is the condition expression, the for loop will stop iterating when this expression returns false. The last component is the post statement, this component is executed after each iteration and is generally used for changing the step counter. You can create a while loop by dropping the first and the last component. If you drop all components you have created an infinite loop.

```
1 // Standard for loop
2 for i := 0; i < max; i++ {
3
4 }
5
6 // While loop
7 for i < max {
8
9 }
10
11 // Infinite loop
12 for {
13
14 }
```

### 3.2.4 Concurrency and channels

It is possible to execute functions concurrent in Go. It uses the idea of threads, but everything is managed by the Go runtime. Because you do not have to manage everything yourself it is easy to use. Goroutines run in the same address space, so shared memory should be synchronized manually, but Go provides packages with useful primitives.

```
1 // A standard function layout
2 func functionName(arguments) {
3
4 }
5
6 // The main function
7 func main() {
8     // New goroutine executes the function
9     go functionName(arguments)
10    // The main execution continues here
11    // concurrent with the goroutine
12 }
```

Apart from the fact that you can execute functions concurrent in a very easy manner with Go, it also has channels. A channel is a first in first out (FIFO) buffer. You can send values to the channel and read from the channel if it is not empty. The channel operator is an arrow (<-) and the data flows in the direction of the arrow. Channels are a good way to let goroutines communicate with each other. If you do not specify a limit for the channel it will be unbuffered, the capacity is zero and thus you are able to synchronize between goroutines. You could also give a second argument, this is the size of the buffer. You can close the channels if you are done or do not need them anymore.

```

1 // Creating an infinite channel
2 channelName := make(chan int)
3 // Creating a limited channel
4 channelName2 := make(chan string, 100)
5 // It is also possible to make a channel from your struct
6 channelName3 := make(chan structType)
7
8 // Sending something on the channel
9 channelName <- something
10 channelName2 <- something
11
12 // Receiving from the channel
13 variable1 := <- channelName
14 variable2 := <- channelName2
15
16 // Closing channels
17 close(channelName)
18 close(channelName2)

```

Sending and receiving are not the only things you can do with a channel. The first functionality I used for the receivers is the range functionality. Range is combined with the for loop. The for loop will continue to receive everything that gets send on the channel, until the channel is closed.

```

1 // Keep listening for values send on the channel
2 for value := range channelName {
3     // Do something with the value
4 }

```

Another functionality I have used is the select statement. I use the select statement to receive from multiple channels at once. A select statement blocks until one of its cases can run, that is the case that will be executed. If more than one cases match, the select statement executes one random from the matched cases.

```

1 // Listening for two channels
2 select {
3 case value := <- channelName:
4     // Received from channelName
5 case value := <- channelName2:
6     // Received from channelName2
7 }

```



## Chapter 4

# Baggage handling system

In this chapter I will explain what a baggage handling system (BHS) is and what it does. I am using the BHS located at Amsterdam Airport Schiphol as an example.

### 4.1 Schiphol

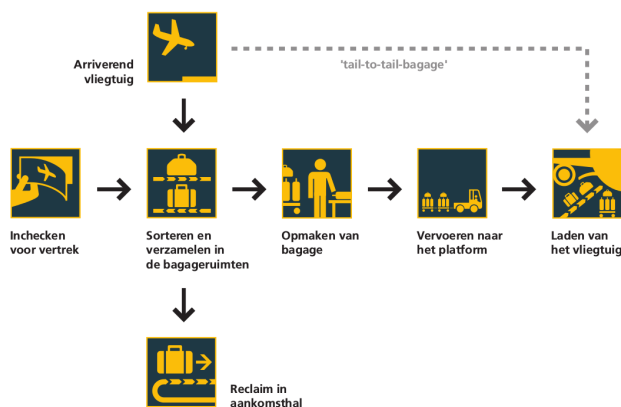
On an annual basis, Schiphol handles over 50 million items of baggage. The BHS, handling all these items, is approximately as large as 26 soccer-fields (129.500 m<sup>2</sup>) and it is possible for a suitcase to travel up to 2.5 kilometres. This area with the BHS comprise a transport system covering over 30 kilometres. The BHS is operated by 110 servers and are powered by almost 10.000 engines. [11]

Schiphol was not built in a day. Starting in 1967, Schiphol built 5 main locations for the baggage to be handled. Because of the year the locations where built in, they contain different technical solutions for transporting the items of baggage from the check-in to the right aeroplane. [4]

### 4.2 My baggage handling system

I have based my program on the technical solutions from nowadays. A system as big as the BHS on Schiphol, has a lot of different components. I left out or simplified some of those components, because they are done fully or partially manual by the personnel or sometimes they do not add a significant value to the program. In the publication by Amsterdam Airport Schiphol you can find a more detailed description about the whole BHS located at Schiphol [4].

Figure 4.1: BHS Amsterdam Airport Schiphol



### 4.2.1 Check-in

The starting point for items of baggage is the check-in. At the check-in new suitcases get a label with all kinds of information. The most important information is the flight code. At Schiphol odd sized items of baggage are processed different from the regular baggage. I do not make this exception in my program, all suitcases are the same. When the item of baggage is labelled it is placed onto a conveyor belt. These conveyor belts transports the items of baggage to underground facilities.

### 4.2.2 Screening

The next components in line are the x-ray-screening machines. Especially nowadays it is very important to scan the items of baggage for unwanted items. I do not have these machines implemented in my program, because I assume that every item of baggage would pass the screening and thus there is no significant value to add this component to my program.

### 4.2.3 Transport and sorting

The next step is transporting the baggage from the x-ray-screening machines to the specific gate where the right aeroplane will depart from. I will distinguish three components, the conveyor belts, the sorting machines and the end stations.

These conveyor belts are straight forward, the conveyor belt connects two other components together. They transport the items of baggage from one end to the other end.

The end station is not so different, it is just a straight conveyor belt rotating at a slow pace. For every aeroplane there is a specific end station and all suitcases that reach the end of their end station are loaded onto small baggage

carts. These carts are driven to the aeroplane and the personnel will load the suitcases into the aeroplane.

The most interesting component are the sorting machines. The sorting machine has to act quickly and as accurate as possible. Multiple scanners scan the barcode on the label of the suitcase, the sorting machine uses that information to determine where the suitcase should go. There is one server in control of the communication. If one of the conveyor belts brakes down the system should react and find another way to transport the items of baggage from point A to point B. Also the system should distribute the load evenly, as it is undesirable that one or only a few conveyor belts transport all the baggage.

#### **4.2.4 Transfer baggage**

Because the flight codes are internationally standardized by the International Air Transport Association (IATA) it is very easy to process transfer baggage [12]. 40% of all the passengers arriving at Amsterdam Airport Schiphol are transfer passengers [13]. To process all these items of baggage as fast as possible, Schiphol created special drop-off points where personnel can unload their baggage carts. From here the transfer baggage already labelled with an IATA flight code follows the same route as suitcases that are checked-in for departing passengers.





# Chapter 5

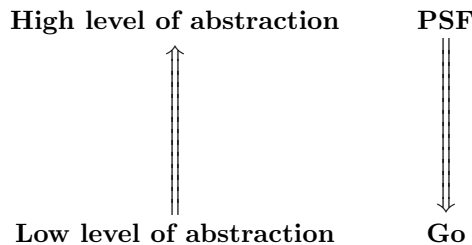
## Implementation

In this chapter I will explain the implementation and the decisions I made during this process.

### 5.1 The process

I divided the process of implementation into two levels of abstraction. I started on an high level of abstraction in PSF. At this level I did not care so much how everything worked, but I designed the communication between the various components. When the specification in PSF was implemented, I started with translating this PSF specification to the target programming language Go on an lower level of abstraction. I found three key points in my process. First I tried to transport a suitcase from point A to point B. Second was sorting the suitcases, but I had the sorting machines choose a random direction. Finally I sorted the suitcases to specific gates, where the right aeroplane was waiting. In the following sections I will describe the code step by step following those key points for each level of abstraction.

Figure 5.1: Abstraction levels of my implementation



### 5.2 PSF

The PSF code is run and tested with the simulator from the Toolkit. To run the simulator I run the following commands.

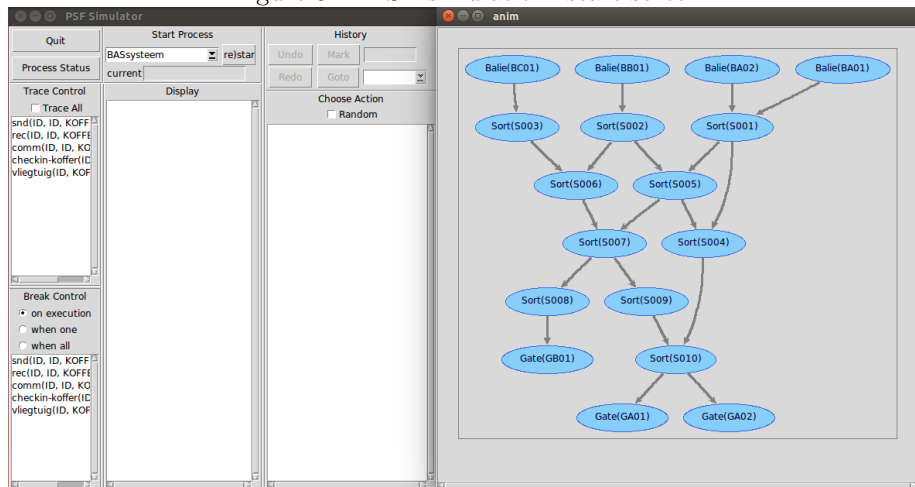
```

$ psf -s BASSystem
$ genanim -N -O BASSystem BASSystem.til
$ tbsim BASSystem.til BASSystem.anim

```

This translates the specification to an animation simulator as seen below.

Figure 5.2: PSF simulation: start screen



To start simulating you have to start the process by pressing the "(re)start" button. You see all the available actions appear on the right. You have three options to execute an action. Option one is clicking the action in the list that just appeared. Option two is using the mouse on the animation, a small list will appear with the available actions of that process. The last option is the small random option, just above the list. Every step a random action from the list is executed. See the next two images for before you execute commands and after you executed a lot of them.

Figure 5.3: PSF simulation: process started

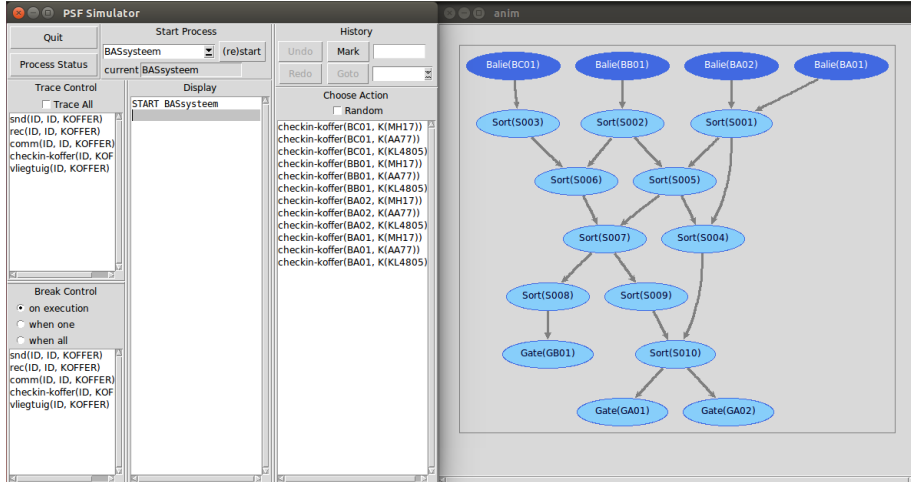
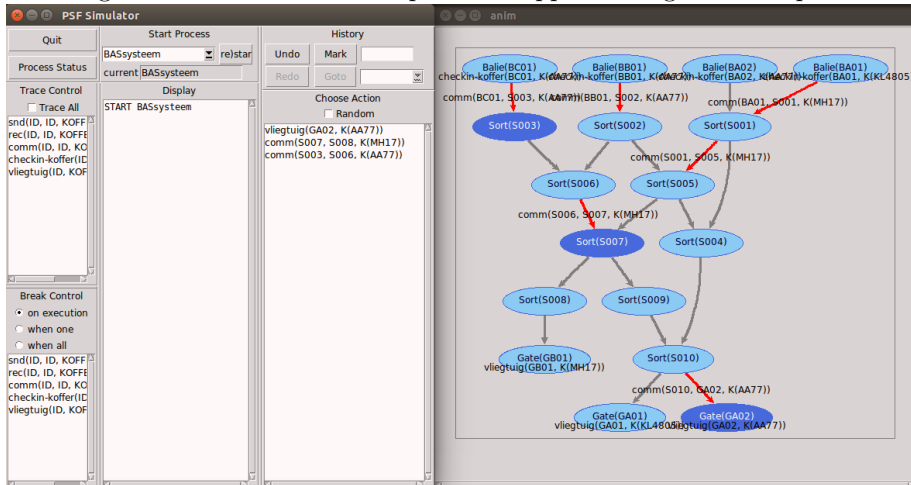


Figure 5.4: PSF simulation: process stopped during random option



### 5.2.1 Step 1: Transporting over a single conveyor belt

The first key point I came up with was transporting a suitcase from a check-in to a gate, over a single conveyor belt. The interesting parts are the definitions of the processes seen below.

The check-in (Balie(bl)) can create suitcases and sends them to conveyor belt "B001". The conveyor belt (Band(bd)) listens for suitcases send to it from the check-in, if it receives a suitcases, the conveyor belt sends it to the gate. The gate (Gate(ga)) listens for suitcases sent from the conveyor belt and if it

receives one it puts it into the aeroplane.

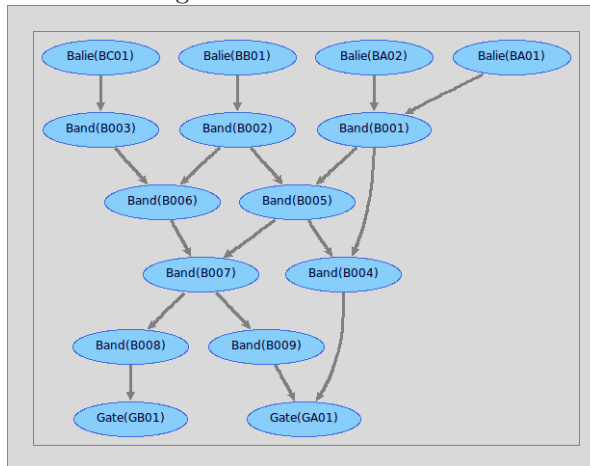
```
1 Balie(bl) =
2   sum(k in KOFFER-set,
3     checkin-koffer(bl, k) .
4     snd(bl, B001, k)
5   ) . Balie(bl)
6
7 Gate(ga) =
8   sum(k in KOFFER-set,
9     sum(bd in BAND-set,
10      rec(bd, ga, k)
11    ) .
12    vliegtuig(ga, k)
13  ) . Gate(ga)
14
15 Band(bd) =
16   sum(k in KOFFER-set,
17     sum(bl in BALIE-set,
18      rec(bl, bd, k)
19    ) .
20     sum(ga in GATE-set,
21      snd(bd, ga, k)
22    )) . Band(bd)
```

Listing 5.1: PSF Step 1: check-in, gate, conveyor belt

## 5.2.2 Step 2: Sorting machines sending in a random direction

Step two was using sorting machines, however they did not really sort, but they send the suitcases in a random direction. The figure on the right gives a better visual of the layout of the system. The code is similar to the code of step one, it is more of the same, as seen in the code snippet below.

Figure 5.5: PSF simulation level 2



I added more check-ins and thus I had to specify what each check-in should do. I created a conditional expression to check which check-in process was running. I specified which check-in should transport its suitcases to which conveyor belts. The conveyor belts now use an alternative composition to choose a random direction for the suitcase it receives. I did not show all the conveyor belts in the code snippet below, because it is just more of the same. The gates are almost the same as in step one, but sometimes they listen for multiple conveyor belts to send them suitcases, that is why I added a generalized alternative composition.

```

1 Balie(b1) =
2   (
3     [b1 = BA01] -> (
4       sum(k in KOFFER-set,
5         checkin-koffer(b1, k) .
6         snd(b1, B001, k)
7       )
8     ) +
9     [b1 = BA02] -> (
10      sum(k in KOFFER-set,
11        checkin-koffer(b1, k) .
12        snd(b1, B001, k)
13      )
14    ) +
15    [b1 = BB01] -> (
16      sum(k in KOFFER-set,
17        checkin-koffer(b1, k) .
18        snd(b1, B002, k)

```

```

19 )
20 ) +
21 [bl = BC01] -> (
22   sum(k in KOFFER-set,
23     checkin-koffer(bl, k) .
24     snd(bl, B003, k)
25   )
26 )
27 ) . Balie(bl)
28
29 Gate(ga) =
30 (
31 [ga = GA01] -> (
32   sum(k in KOFFER-set,
33     sum(id in rec-set-GA01,
34       rec(id, ga, k)
35     ) .
36     vliegtuig(ga, k)
37   )
38 ) +
39 [ga = GB01] -> (
40   sum(k in KOFFER-set,
41     rec(B008, ga, k) .
42     vliegtuig(ga, k)
43   )
44 )
45 ) . Gate(ga)
46
47 Band(bd) =
48 (
49 [bd = B001] -> (
50   sum(k in KOFFER-set,
51     sum(id in rec-set-B001,
52       rec(id, bd, k)
53     ) .
54     ( snd(bd, B004, k) + snd(bd, B005, k) )
55   )
56 ) +
57 (Ditto for the rest.)
58 +
59 [bd = B009] -> (
60   sum(k in KOFFER-set,
61     rec(B007, bd, k) .
62     snd(bd, GA01, k)
63   )
64 )
65 ) . Band(bd)

```

Listing 5.2: PSF Step 2: check-in, gate, conveyor belt

### 5.2.3 Step 3: Sorting machines sending in a specified direction

With step three of the process I gave the sorting machines a condition to sort the suitcases. I had to make some adjustments and while I was doing that I also changed some names I used to project a better simulation of the BHS. I also changed the layout of the system, as you can see on the right.

First I made sure every suitcase got a label.

This label contains the flight code of the aeroplane the suitcase belongs to. It is important that the system knows which aeroplane is located at which gate number. To match aeroplanes with gate numbers I created a lookup table. A lookup table is implemented as a dictionary with the flight code as key and the gate number as value. The code snippet below shows how I implemented this.

```

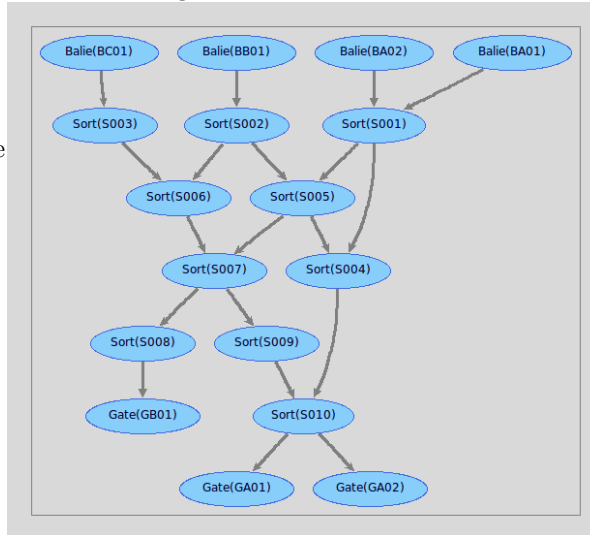
1 functions
2   K : LABEL -> KOFFER
3   KL4805 : -> LABEL
4   AA77 : -> LABEL
5   MH17 : -> LABEL
6   vertrek-gate : LABEL -> ID
7   get-label : KOFFER -> LABEL
8 variables
9   l : -> LABEL
10 equations
11 [101] get-label(K(l)) = l
12 [201] vertrek-gate(KL4805) = GA01
13 [202] vertrek-gate(AA77) = GA02
14 [203] vertrek-gate(MH17) = GB01

```

Listing 5.3: PSF Step 3: Lookup table

I also made some structural decisions this step. I made a definition per check-in process, so no more conditional expressions to check what process is running. The rest of the check-in definitions remained the same. The gates had

Figure 5.6: PSF simulation level 3



the same makeover as the check-ins, they now have a definition for each process individually. The conveyor belts are now named "Sort", because that is a more suitable name. The sorting machines use the "get-label" equation to read the label from the suitcase and they use the "vertrek-gate" equation to look up what the direction for that label is. The code snippet below shows these changes.

```

1 Balie(BA01) =
2   sum(k in KOFFER-set,
3     checkin-koffer(BA01, k) .
4     snd(BA01, S001, k)
5   ) . Balie(BA01)
6 Balie(BA02) =
7   sum(k in KOFFER-set,
8     checkin-koffer(BA02, k) .
9     snd(BA02, S001, k)
10  ) . Balie(BA02)
11 Balie(BB01) =
12  sum(k in KOFFER-set,
13    checkin-koffer(BB01, k) .
14    snd(BB01, S002, k)
15  ) . Balie(BB01)
16 Balie(BC01) =
17  sum(k in KOFFER-set,
18    checkin-koffer(BC01, k) .
19    snd(BC01, S003, k)
20  ) . Balie(BC01)
21
22 Gate(GA01) =
23  sum(k in KOFFER-set,
24    rec(S010, GA01, k) .
25    vliegtuig(GA01, k)
26  ) . Gate(GA01)
27 Gate(GA02) =
28  sum(k in KOFFER-set,
29    rec(S010, GA02, k) .
30    vliegtuig(GA02, k)
31  ) . Gate(GA02)
32 Gate(GB01) =
33  sum(k in KOFFER-set,
34    rec(S008, GB01, k) .
35    vliegtuig(GB01, k)
36  ) . Gate(GB01)
37
38 Sort(S001) =
39  sum(k in KOFFER-set,
40    sum(id in rec-set-S001,
41      rec(id, S001, k)
42    ) .
43    (
44      [vertrek-gate(get-label(k)) = GA01] -> (
45        snd(S001, S004, k)
46      ) +
47      [vertrek-gate(get-label(k)) = GA02] -> (
48        snd(S001, S004, k)
49      ) +
50      [vertrek-gate(get-label(k)) = GB01] -> (
51        snd(S001, S005, k)

```



```

52 )
53 )
54 ) . Sort(S001)
55 (Ditto for the rest.)
56 Sort(S010) =
57   sum(k in KOFFER-set,
58     sum(id in rec-set-S010,
59       rec(id, S010, k)
60     ) .
61   (
62     [vertrek-gate(get-label(k)) = GA01] -> (
63       snd(S010, GA01, k)
64     ) +
65     [vertrek-gate(get-label(k)) = GA02] -> (
66       snd(S010, GA02, k)
67     ) +
68     delta
69   )
70 ) . Sort(S010)

```

Listing 5.4: PSF Step 3: check-in, gate, conveyor belt

### 5.3 Go

On the website of Go there is a tutorial to install Go [14]. After the installation it is possible to use a new command in your terminal to compile Go programs.

```
$ go install github.com/user/program
```

When your program is compiled, it creates an executable file in the bin directory of your workspace. When you execute the file, the results are printed in the terminal. I let the program print out events when suitcases reach a gate. I also built in an option to terminate the execution, by pressing "Enter". Upon terminating successful I let the program print out some numbers I have collected during the execution, about the amount of suitcases checked-in and delivered to a certain gate. Below are some images displaying the execution process.

Figure 5.7: Go execute program: How to execute the file

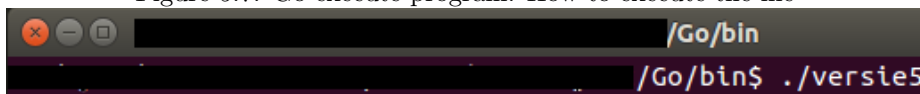
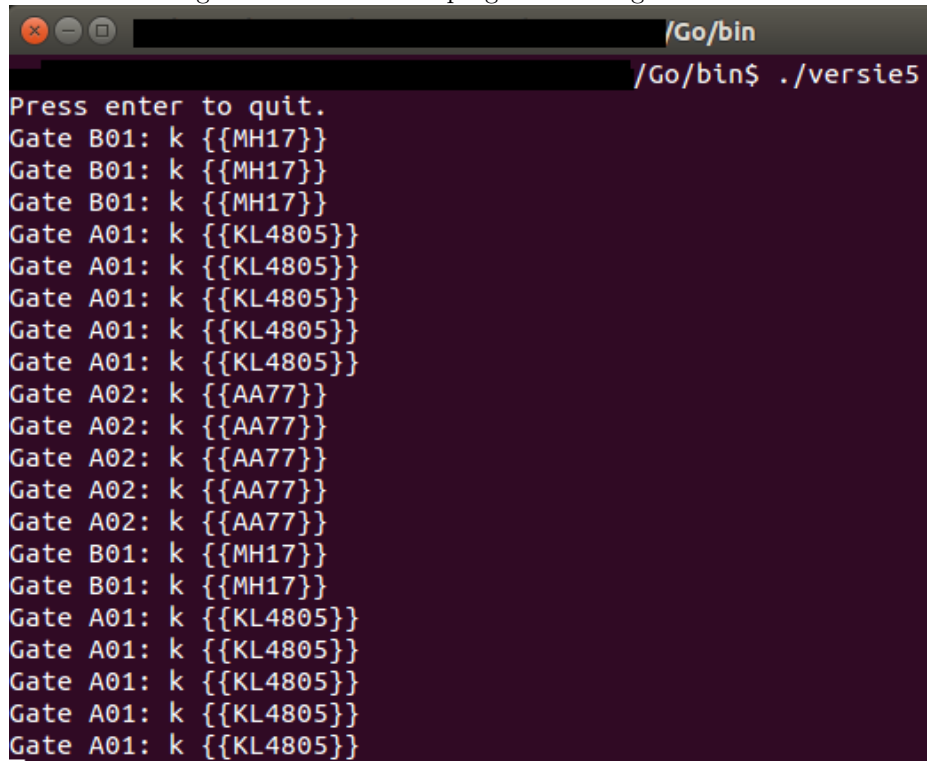


Figure 5.8: Go execute program: Begin of execution

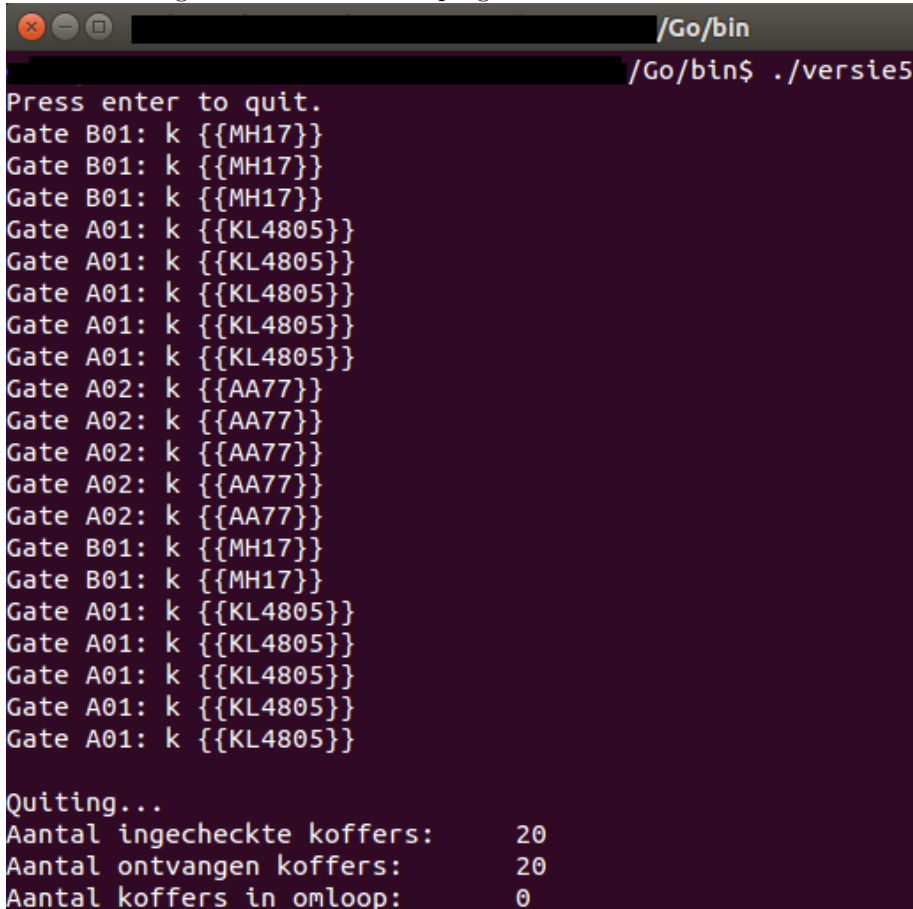


Figure 5.9: Go execute program: During execution

A terminal window with a dark background and light text. The title bar shows window control icons and the path "/Go/bin". The prompt is "/Go/bin\$ ./versie5". The output consists of a series of lines: "Press enter to quit.", followed by three "Gate B01: k {{MH17}}", then four "Gate A01: k {{KL4805}}", then five "Gate A02: k {{AA77}}", then two "Gate B01: k {{MH17}}", and finally four "Gate A01: k {{KL4805}}".

```
/Go/bin
/Go/bin$ ./versie5
Press enter to quit.
Gate B01: k {{MH17}}
Gate B01: k {{MH17}}
Gate B01: k {{MH17}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A02: k {{AA77}}
Gate A02: k {{AA77}}
Gate A02: k {{AA77}}
Gate A02: k {{AA77}}
Gate A02: k {{AA77}}
Gate B01: k {{MH17}}
Gate B01: k {{MH17}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
```

Figure 5.10: Go execute program: After the execution



```
/Go/bin
/Go/bin$ ./versie5
Press enter to quit.
Gate B01: k {{MH17}}
Gate B01: k {{MH17}}
Gate B01: k {{MH17}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A02: k {{AA77}}
Gate A02: k {{AA77}}
Gate A02: k {{AA77}}
Gate A02: k {{AA77}}
Gate A02: k {{AA77}}
Gate B01: k {{MH17}}
Gate B01: k {{MH17}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Gate A01: k {{KL4805}}
Quiting...
Aantal ingecheckte koffers:    20
Aantal ontvangen koffers:    20
Aantal koffers in omloop:    0
```

### 5.3.1 Step 1: Transporting over a single conveyor belt

Go can create concurrency with goroutines. A special function-call "go" creates a new process executing the function it is combined with. In Go the first step was creating the same processes that I had in PSF. In Go I also had to make some channels to communicate between goroutines, these channels are the conveyor belts in my program. I added an extra variable to the check-in function, this "aantalKoffers" is an integer containing the amount of suitcases being checked-in. To create the suitcases I use a for-loop. Like you can see in the code snippet below.

Go is executed in the terminal and can print results. I let the program print some numbers I collected. You can see I have two variables, "koffersSnd" and "koffersRec", they contain the number of suitcases send and number of suitcases received. It is good to know whether indeed all your send suitcases

are also received.

```
1 func incheckbalieA01(B001 chan <- Koffer, aantalKoffers int) {
2   for i := 0; i < aantalKoffers; i++ {
3     k := Koffer{
4       number: i
5     }
6     B001 <- k
7     koffersSnd++
8   }
9 }
10
11 func gateA01(B002 <- chan Koffer) {
12   for k := range B002 {
13     fmt.Println("Gate A01: k", k)
14     KoffersRec++
15   }
16 }
17
18 func sort001(B001 <- chan Koffer, B002 chan <- Koffer) {
19   for {
20     k := <- B001
21     B002 <- k
22   }
23 }
```

Listing 5.5: Go Step 1: check-in, gate, conveyor belt

### 5.3.2 Step 2: Sorting machines sending in a random direction

For the second step I used the same system layout of PSF step two.

I used a function called "rand.Intn()" to create a random number for the sorting machine. In this step I also generalised the sending and receiving channels as function arguments. I renamed them, so I could easily add more functions. The complexity in the code of the sorting machine is determined by the amount of receiving channels and the amount of outgoing channels.

I let gate function print a statement every time it receives a suitcase. Printing this information is useful for following the flow of the suitcases. I decided not to print the suitcases in the sorting machines, because the amount of printing statements would be too much information in the terminal.

```
1 func incheckbalieA01(snd1 chan <- Koffer, aantalKoffers int) {
2   for i := 0; i < aantalKoffers; i++ {
3     k := Koffer{
4       number: i
5     }
6     snd1 <- k
7     koffersSnd++
8   }
9 }
10
11 func incheckbalieA02(snd1 chan <- Koffer, aantalKoffers int) {
12   for i := 0; i < aantalKoffers; i++ {
13     k := Koffer{
```

```

13     number: i
14     }
15     snd1 <- k
16     koffersSnd++
17 }
18 }
19 func incheckbalieB01(snd1 chan <- Koffer, aantalKoffers int) {
20     for i := 0; i < aantalKoffers; i++ {
21         k := Koffer{
22             number: i
23         }
24         snd1 <- k
25         koffersSnd++
26     }
27 }
28 func incheckbalieC01(snd1 chan <- Koffer, aantalKoffers int) {
29     for i := 0; i < aantalKoffers; i++ {
30         k := Koffer{
31             number: i
32         }
33         snd1 <- k
34         koffersSnd++
35     }
36 }
37
38 func gateA01(rec <-chan Koffer) {
39     for k := range rec {
40         fmt.Println("Gate A01: k", k)
41         KoffersRec++
42     }
43 }
44 func gateB01(rec <-chan Koffer) {
45     for k := range rec {
46         fmt.Println("Gate B01: k", k)
47         KoffersRec++
48     }
49 }
50
51 func sort001(rec1 <- chan Koffer, rec2 <- chan Koffer, snd1 chan <-
52     Koffer, snd2 chan <- Koffer) {
53     for {
54         select {
55             case k := <- rec1:
56                 if rand.Intn(2) == 1 {
57                     snd1 <- k
58                 } else {
59                     snd2 <- k
60                 }
61             case k := <- rec2:
62                 if rand.Intn(2) == 1 {
63                     snd1 <- k
64                 } else {
65                     snd2 <- k
66                 }
67         }
68     }

```

```

69 (Ditto for the rest.)
70 func sort009(rec1 <- chan Koffer, snd1 chan <- Koffer) {
71     for {
72         k := <- rec1
73         snd1 <- k
74     }
75 }

```

Listing 5.6: Go Step 2: check-in, gate, conveyor belt

### 5.3.3 Step 3: Sorting machines sending in a specified direction

For step three in Go I also used a lookup table. It is the same idea as with PSF to create a dictionary with the flight code as key and the gate number as value. In go I have implemented this like below.

```

1 type Label struct {
2     vluchtnummer string
3 }
4
5 type Koffer struct {
6     label Label
7 }
8
9 var vluchtnummerVertaler = map[string]string{}
10
11 vluchtnummerVertaler["KL4805"] = "GA01"
12 vluchtnummerVertaler["AA77"] = "GA02"
13 vluchtnummerVertaler["MHL7"] = "GB01"

```

Listing 5.7: Go Step 3: Lookup table

Because I already had the same names as in PSF step three, the changes I made this step where making sure the sorting machines sorted with the new lookup table. I made sure the check-ins are giving the suitcases labels and I used the same "select/case" function as in the second step for the sorting machines to listen to multiple channels. Below is the code snippet of the Go program during the final step.

To create suitcases I use a nested structure. I have two structures the "Koffer" structure and the "Label" structure, the "Koffer" structure contains a variable (label) and that variable is of the type "Label", so that is the nested structure. The "Label" structure contains a string with the IATA flight code of the aeroplane.

```

1 func incheckbalieA01(snd chan<- Koffer, IATA string, aantalKoffers
2     int) {
3     for i := 0; i < aantalKoffers; i++ {
4         k := Koffer{
5             label: Label{
6                 vluchtnummer: IATA,
7             },
8         },
9     }
10 }

```

```

8     snd <- k
9     koffersSnd++
10  }
11 }
12 func incheckbalieA02(snd chan<- Koffer, IATA string, aantalKoffers
13 int) {
14     for i := 0; i < aantalKoffers; i++ {
15         k := Koffer{
16             label: Label{
17                 vluchtnummer: IATA,
18             },
19         }
20         snd <- k
21         koffersSnd++
22     }
23 }
24 func incheckbalieB01(snd chan<- Koffer, IATA string, aantalKoffers
25 int) {
26     for i := 0; i < aantalKoffers; i++ {
27         k := Koffer{
28             label: Label{
29                 vluchtnummer: IATA,
30             },
31         }
32         snd <- k
33         koffersSnd++
34     }
35 }
36 func incheckbalieC01(snd chan<- Koffer, IATA string, aantalKoffers
37 int) {
38     for i := 0; i < aantalKoffers; i++ {
39         k := Koffer{
40             label: Label{
41                 vluchtnummer: IATA,
42             },
43         }
44         snd <- k
45         koffersSnd++
46     }
47 }
48 func gateA01(rec <-chan Koffer) {
49     for k := range rec {
50         fmt.Println("Gate A01: k", k)
51         KoffersRec++
52     }
53 }
54 func gateA02(rec <-chan Koffer) {
55     for k := range rec {
56         fmt.Println("Gate A02: k", k)
57         KoffersRec++
58     }
59 }
60 func gateB01(rec <-chan Koffer) {
61     for k := range rec {

```

```

62     fmt.Println(" Gate B01: k", k)
63     KoffersRec++
64 }
65 }
66
67 func sort001(rec1 <-chan Koffer, rec2 <-chan Koffer, snd1 chan<-
68     Koffer, snd2 chan<- Koffer) {
69     for {
70         select {
71         case k := <-rec1:
72             vertaald := vluchtnummerVertaler[k.label.vluchtnummer]
73             if vertaald == "GA01" {
74                 snd2 <- k
75             } else if vertaald == "GA02" {
76                 snd2 <- k
77             } else if vertaald == "GB01" {
78                 snd1 <- k
79             }
80         case k := <-rec2:
81             vertaald := vluchtnummerVertaler[k.label.vluchtnummer]
82             if vertaald == "GA01" {
83                 snd2 <- k
84             } else if vertaald == "GA02" {
85                 snd2 <- k
86             } else if vertaald == "GB01" {
87                 snd1 <- k
88             }
89         }
90     }
91     (Ditto for the rest.)
92     func sort010(rec1 <-chan Koffer, rec2 <-chan Koffer, snd1 chan<-
93         Koffer, snd2 chan<- Koffer) {
94         for {
95             select {
96             case k := <-rec1:
97                 vertaald := vluchtnummerVertaler[k.label.vluchtnummer]
98                 if vertaald == "GA01" {
99                     snd1 <- k
100                 } else if vertaald == "GA02" {
101                     snd2 <- k
102                 }
103             case k := <-rec2:
104                 vertaald := vluchtnummerVertaler[k.label.vluchtnummer]
105                 if vertaald == "GA01" {
106                     snd1 <- k
107                 } else if vertaald == "GA02" {
108                     snd2 <- k
109                 }
110             }
111         }

```

Listing 5.8: Go Step 3: check-in, gate, conveyor belt



# Chapter 6

## Results

In this chapter I will discuss several results I found during my implementation of the BHS in PSF and Go.

### 6.1 Processes

All the different components in the BHS are all processes in PSF. In Go I had to create the same components on a concurrent level as in PSF. I found that for every process in PSF I made a function in Go. These function are executed on a concurrent level.

```
1 Balie(BA01) =
2 Balie(BA02) =
3 Balie(BB01) =
4 Balie(BC01) =
5 Gate(GA01) =
6 Gate(GA02) =
7 Gate(GB01) =
8 Sort(S001) =
9 (Ditto for the rest.)
10 Sort(S010) =
```

Listing 6.1: PSF processes

```
1 func incheckbalieA01(snd chan<- Koffer,
   IATA string, aantalKoffers int) {
2 func incheckbalieA02(snd chan<- Koffer,
   IATA string, aantalKoffers int) {
3 func incheckbalieB01(snd chan<- Koffer,
   IATA string, aantalKoffers int) {
4 func incheckbalieC01(snd chan<- Koffer,
   IATA string, aantalKoffers int) {
5 func gateA01(rec <-chan Koffer) {
6 func gateA02(rec <-chan Koffer) {
7 func gateB01(rec <-chan Koffer) {
8 func sort001(rec1 <-chan Koffer, rec2
   <-chan Koffer, snd1 chan<- Koffer, snd2
   chan<- Koffer) {
9 (Ditto for the rest.)
10 func sort010(rec1 <-chan Koffer, rec2
   <-chan Koffer, snd1 chan<- Koffer, snd2
   chan<- Koffer) {
```

Listing 6.2: GO processes

## 6.2 Communication

In PSF the communication between processes is defined by the communication section and the encapsulation in the processes. In Go you can communicate between goroutines with channels, so for every process pair you want to communicate between you should create a channel.

```
1 sets
2   of atoms
3     H = { snd(id1, id2, d),
4           rec(id1, id2, d) |
5             id1 in ID, id2 in
6             ID, d in KOFFER}
7   communications
8     snd(id1, id2, d) | rec(id1,
9       id2, d) = comm(id1, id2, d)
10  for
11    id1 in ID, id2 in ID, d
12    in KOFFER
13  definitions
14    BASsystem = encaps(H,
15      Balies || Gates || Sorters)
```

Listing 6.3: PSF communication

```
1 B001 := make(chan Koffer,
2   incheckbaliebuffer)
3 B002 := make(chan Koffer,
4   incheckbaliebuffer)
5 B003 := make(chan Koffer,
6   incheckbaliebuffer)
7 B004 := make(chan Koffer,
8   incheckbaliebuffer)
9 B005 := make(chan Koffer,
10  sortbuffer)
11 (Ditto for the rest.)
12 B018 := make(chan Koffer,
13  gatebuffer)
14 B019 := make(chan Koffer,
15  gatebuffer)
```

Listing 6.4: GO communication

## 6.3 Equations

If you look at the equations section in the PSF specification, you will see two types of equations. The two types of equations are the "get-label" equation and the "vertrek-gate" equation. There are multiple ways to translate the PSF equations to Go. In PSF they are both equations but if you look at the translation to Go, you will see a structure and a dictionary. For the translation of the equation section in PSF to Go, you need to have knowledge about the PSF specification. You should know the purpose of the equations, before you can actually translate them to Go.

The "get-label" equation in PSF is used to get the label from a suitcase and is used by the sorting machines. In Go we defined the suitcase and label as structures, therefore we can use the built-in functionality of Go's structures to get the label.

```

1 functions
2 K : LABEL -> KOFFER
3 KL4805 : -> LABEL
4 AA77 : -> LABEL
5 MH17 : -> LABEL
6 get-label : KOFFER -> LABEL
7 variables
8 l : -> LABEL
9 equations
10 [101] get-label(K(l)) = l

```

Listing 6.5: PSF equations: get-label

```

1 type Label struct {
2     vluchtnummer string
3 }
4
5 type Koffer struct {
6     label Label
7 }
8
9 k := Koffer{
10     label: Label{
11         vluchtnummer: IATA,
12     },
13 }
14
15 flightCode := k.label.
    vluchtnummer

```

Listing 6.6: GO equations

The second equation, the "vertrek-gate" equation, has the same functionality as a lookup table. The PSF specification uses term rewriting method for its equations, so when it encounters the left hand side of the equation, it is replaced with the right hand side of the equation. In Go I used a dictionary for the same result. A dictionary stores keys with values in a map, like in the code snippet below.

```

1 functions
2 K : LABEL -> KOFFER
3 KL4805 : -> LABEL
4 AA77 : -> LABEL
5 MH17 : -> LABEL
6 GA01 : -> ID
7 GA02 : -> ID
8 GB01 : -> ID
9 vertrek-gate : LABEL -> ID
10 equations
11 [201] vertrek-gate(KL4805) =
    GA01
12 [202] vertrek-gate(AA77) =
    GA02
13 [203] vertrek-gate(MH17) =
    GB01

```

Listing 6.7: PSF equations

```

1 var vluchtnummerVertaler = map[
2     string] string {}
3 vluchtnummerVertaler["KL4805"] =
4     "GA01"
5 vluchtnummerVertaler["AA77"] = "
    GA02"
6 vluchtnummerVertaler["MH17"] = "
    GB01"

```

Listing 6.8: GO equations



## Chapter 7

# Conclusion

The purpose of this research was engineering a software application with process algebra with the programming language Go as target. I decided I would use the BHS located at Amsterdam Airport Schiphol as an example.

I gained knowledge about PSF and Go to implement this BHS on different levels of abstraction. To describe the systems behaviour on a high level of abstraction I used PSF. Go is useful to implement the behaviour of the system on a low level of abstraction, because Go has a built-in functionality to execute functions on a concurrent level. It is possible to communicate with channels over these goroutines. These functionalities combined makes Go a good target for the implementation of the PSF specification.

I implemented the BHS in three steps. Step one is transporting items of baggage from point A to point B over a single conveyor belt. During the second step I implemented sorting machines that would sort the suitcases in a random direction. The last step was to make sure the sorting machines sorted the suitcases in the right direction of the right aeroplane.

I found three main translation similarities from PSF to Go. I found that processes in PSF could be translated to functions in Go, if you would execute those functions on a concurrent level with goroutines. I also found that the communication between different processes could be achieved with channels in Go. The last thing I found was that with some knowledge of the PSF specification, you could translate equations in different ways. The two ways I needed was to implement a lookup table as a dictionary and I used a structure in Go to represent a suitcase, where you could store a label in with the flight code.



# Bibliography

- [1] Se-pa home page. <https://staff.fnwi.uva.nl/b.diertens/se-pa/>.
- [2] Sjouke Mauw. *PSF - A Process Specification Formalism*. PhD thesis, Universiteit van Amsterdam, 1991.
- [3] Psf home page. <https://staff.fnwi.uva.nl/b.diertens/psf/>.
- [4] Amsterdam Airport Schiphol. Bagage op schiphol.
- [5] Frequently asked questions go. <https://golang.org/doc/faq>.
- [6] Go release history. <https://golang.org/project/>.
- [7] About the go command. [https://golang.org/doc/articles/go\\_command.html](https://golang.org/doc/articles/go_command.html).
- [8] Rob Pike. Go at google: Language design in the service of software engineering. <https://talks.golang.org/2012/splash.article>, 2012.
- [9] Documentations go. <https://golang.org/doc/>.
- [10] Mark Summerfield. *Programming in Go: Creating Applications for the 21st Century*. Addison-Wesley, 2012.
- [11] Amsterdam Airport Schiphol. Baggage at schiphol, 2016.
- [12] Iata codes. <http://www.iata.org/services/pages/codes.aspx>.
- [13] Amsterdam Airport Schiphol. Facts and figures, 2015.
- [14] Go installation. <https://golang.org/doc/install>.
- [15] Bob Diertens. *Software Engineering with Process Algebra*. PhD thesis, Universiteit van Amsterdam, 2009.
- [16] Daan Staudt. A case study in software engineering with psf: A domotics application. Technical report, University of Amsterdam, Programming Research Group, 2008.

- [17] Bob Diertens. Software engineering with process algebra: Modelling client / server architectures. Technical report, University of Amsterdam, Programming Research Group, 2009.
- [18] Amsterdam Airport Schiphol. Traffic review, 2015.