Discrete event simulation in Java with the use of frameworks

Verhoef, Céline

Award date:
2017

Discrete event simulation in Java with the use of frameworks

Céline Verhoef (0896270)
supervisor: Marko Boon

Bachelor Final Project

May 18, 2017

Abstract

Frameworks are developed in the Java programming language in order to support the implementation of discrete event simulations. In this report, we explore the pros and cons of programming with the use of frameworks. We consider the frameworks that are recently updated and support the process-interaction approach. We compare the implementations based on several criteria and conclude the report with a recommendation for which implementation is the best to use.

Contents

1 Introduction **4**

2 Discrete event simulation **5**

3 Models **6**
3.1 M/M/c tandem queue 6
3.2 Polling system with gated service 7
3.3 Discrete-time fixed-cycle traffic light 8

4 Implementation without the use of frameworks **10**
4.1 M/M/c tandem queue 10
4.2 Polling system with gated service 14
4.3 Discrete-time fixed-cycle traffic light 17
4.4 Improvements 20

5 Choice of frameworks **23**
5.1 Selection 23
5.2 Description of DESMO-J 24
5.3 Description of SSJ 25

6 Implementation with the use of frameworks **27**
6.1 DESMO-J 27
6.1.1 M/M/c tandem queue 27
6.1.2 Polling system with gated service 30
6.1.3 Discrete-time fixed-cycle traffic light 32
6.1.4 Output 33
6.2 SSJ 34
6.2.1 M/M/c tandem queue 34
6.2.2 Polling system with gated service 35
6.2.3 Discrete-time fixed-cycle traffic light 37
6.2.4 Output 39

7 Results **41**

8 Conclusion **47**

Appendix **49**
A Performance Results 49

Chapter 1

Introduction

A discrete event system is a system which is completely determined by stochastic events. When an event happens, it causes a change in the state of the system. Between these events the state of the system will only change according to some deterministic pattern. We are interested in performance measures for these types of systems. For complex systems, closed form formulas for the performance measures are not always available. In order to analyse these type of systems simulation is used.

There are numerous tools developed to provide the means for discrete event simulation. Tools such as Arena [1] provide a "drag-and-drop" interface and can therefore be used by people without any programming experience. However, these kinds of tools do not offer the functionality to simulate all complex systems. A general purpose programming language such as Java does provide all flexibility that is needed. Since Java is very suitable for simulation purposes, numerous frameworks are developed to decrease the implementation effort and to make the resulting code more understandable. Especially complex systems result in complex code. For such systems frameworks could be helpful. In this project we investigate the pros and cons of using a framework and which framework is the best based on the criteria performance, understandability, implementation effort, scalability, reliability and data collection.

By means of three queuing models we investigate the advantages and disadvantages of the implementations with and without frameworks. The models are an M/M/c tandem queue, a polling system with gated service and a discrete-time fixed-cycle traffic light. After having described each model in Chapter 3, we explain in Chapter 4 the implementation details of the programs without using a framework. These implementations provide a baseline for the performance of the simulation programs. In Chapter 5, we select and describe two frameworks that seem to offer the best functionality: DESMO-J and SSJ. We explain the implementation details of the same models with the use of these frameworks in Chapter 6. At last, the implementations are compared based on several criteria in Chapter 7 and a conclusion is given in Chapter 8.

In the literature numerous papers are written about discrete event simulation and also specific about the frameworks that can be used for that. However, the terms used in these papers vary between framework, library and toolkit. To avoid confusion, we will use in this report only the term framework.

Chapter 2

Discrete event simulation

A discrete event simulation can be written from different viewpoints. One viewpoint is to see the system as a sequence of stochastic events that causes changes in the system. The other viewpoint is to see the entities of the system as processes that have a certain behaviour during the simulation. These viewpoints result in two different approaches for implementing a simulation: the event-scheduling approach and the process-interaction approach. In general the event-based approach results in a faster implementation, while the process-based approach is more understandable. In this chapter we explain the use of these two approaches using an M/M/1 queueing model as main example.

In the event-scheduling approach the model is seen as a collection of events. When an event occurs, the state of the systems change. In the M/M/1 queueing model multiple types of events occur. These events are the arrival and departure of a customer, the server taking a customer into service and the server finishing serving a customer. The events that a server is finished serving a customer and that the customer departs coincide. Also the arrival of a customer and the server taking that customer into service may coincide. In order to handle the events a Future Event Set (FES) should be created. Within this set the events are listed sorted on time of occurrence. When the simulation is running, each time the next scheduled event is handled.

The process-interaction approach has a more intuitive way of programming than the event-scheduling approach. However, the process-interaction is more difficult to implement. This difficulty is the cause that many frameworks are developed in order to increase the understandability of the simulation programs. In this approach the model is seen as a collection of processes. These processes describe the behaviour of entities during the simulation. In the M/M/1 queueing model two types of processes can be distinguished: the customer process and the server process. There is one server process during the course of one simulation, while multiple customer processes start and end before the end of the simulation. The entities in the system interact with each other, implying that the processes need to be synchronized.

In this project the event-scheduling approach is used to implement the models without a framework. For the implementation with the use of frameworks the process-interaction approach is used. The frameworks provide the means of describing the behaviour of processes and the way they interact with each other.

Chapter 3

Models

In this chapter we describe the three models we implemented and used to compare the different implementations. The models are an M/M/c tandem queue, a polling system with gated service and a discrete-time fixed-cycle traffic light. Each of these models can also be analysed theoretically. The results of the theoretical analysis can be compared with the results obtained in the simulations to verify the correctness of the implementations.

3.1 M/M/c tandem queue

The tandem queue is a queueing network that consists of multiple queues and serving stations lined up one behind the other. The departure from one station results in an arrival in the next queue. Figure 3.1 depicts a schematic representation of this model. A customer enters the system in the first queue and leaves the system after he is served at the last station.

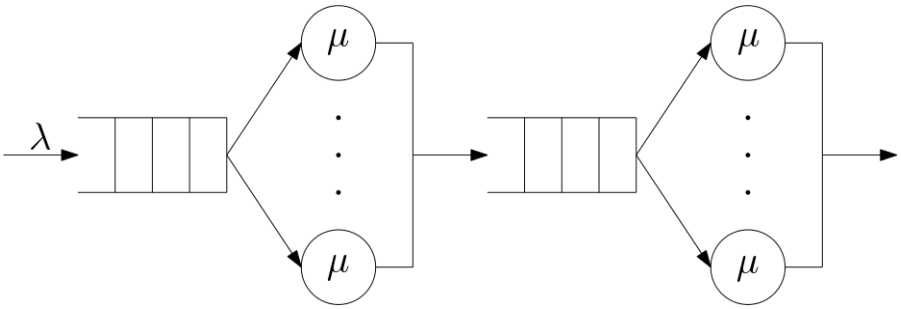

Figure 3.1: M/M/c tandem queue

We consider an M/M/c tandem queue, which means that at each station there are c servers. We assume that the service times are exponentially distributed and that all servers are equal. The arrival of customers in the system is assumed to follow a Poisson distribution. We also assume that there is no travel time between a station and the next queue.

The parameters in this model are:

- the arrival rate λ;
- the service rate μ;

- the number of stations N;
- and the number of servers per station c.

In order for the system to be stable it should hold that $\rho = \frac{\lambda}{c\mu} < 1$. If this condition is violated, more customers arrive than the servers can handle.

The performance measures of interest are:

- the mean waiting time per station;
- and the mean sojourn time.

The arrival of customers follows a Poisson distribution. A remarkable result for this system, is that the departure process from each station is also a Poisson process, similar to the arrival process. The arrival rate remains constant for all stations. This implies that the mean waiting time is the same at each station. A closed-form formula for the mean waiting time is described in the lecture notes of Adan and Resing [2]. It states that the mean waiting time is given by:

$$\mathbb{E}[W] = \frac{(c\rho)^c}{c!}\left((1-\rho)\sum_{n=0}^{c-1}\frac{(c\rho)^n}{n!} + \frac{(c\rho)^c}{c!}\right)^{-1} \cdot \frac{1}{1-\rho}\cdot\frac{1}{c\mu}. \tag{3.1}$$

Since the service time is exponentially distributed with parameter μ, the mean service time is given by $\frac{1}{\mu}$. To obtain the mean sojourn time we simply have to add the mean waiting time and mean service time and multiply it by the number of stations. So, the mean sojourn time is given by:

$$E[S] = N(\mathbb{E}[W] + \frac{1}{\mu}). \tag{3.2}$$

3.2 Polling system with gated service

In a polling system there are multiple queues and only one server. The server serves one queue at the time. After the server is finished serving one queue, he switches to the next one. When the server has served all queues, the server starts again with serving the first queue. Figure 3.2 shows a sketch of this model. The gated service means that only the customers that are in the queue the moment the server arrives will be served that round.

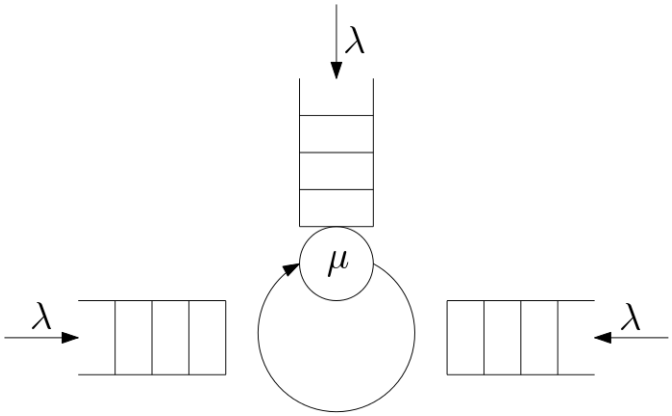

Figure 3.2: Polling system

We assume again that the service time is exponentially distributed and that the arrival of customers follows a Poisson distribution. The arrival of customers in a queue is independent of the arrivals in the other queues. The time that the server needs to switch between the queues is called the switch-over time. We assume that the switch-over time is constant. We consider a symmetric system, which means that the server has always the same service rate and switch-over time and customers arrive at the same rate at each queue.

The parameters in this model are:

- the arrival rate λ;
- the service rate μ;
- the switch-over time s;
- and the number of stations N;

In order for the system to be stable it should hold that $\rho = \frac{N\lambda}{\mu} < 1$.

The performance measures of interest are:

- the mean waiting time per station;
- and the mean cycle time of the server.

For these performance measures closed form formulas are derived by Boxma and Groenendijk [3]. The formulas described in this paper apply to more general and asymmetric polling systems. Since we consider a symmetric polling system, we can simplify these formulas. This results in the following formula for the mean waiting time per station:

$$\mathbb{E}[W] = \frac{\rho}{1-\rho}\left(\frac{1}{\mu} + s\right) + \frac{Ns}{2} + \frac{Ns\rho}{2(1-\rho)}\left(1 - \frac{1}{N}\right). \tag{3.3}$$

The formula to compute the mean cycle time becomes:

$$\mathbb{E}[C] = \frac{Ns}{1-\rho}. \tag{3.4}$$

3.3 Discrete-time fixed-cycle traffic light

In this model we consider a single traffic light. The time in the model is discrete, which means that the simulation time is divided into equal length time slots. In each time slot the traffic light is either red or green. It is a fixed-cycle traffic light, which means that for a specified number of time slots the light is repeatedly green and red. Independent of the time slots cars arrive at the traffic light. We assume that the arrival of cars follows a Poisson distribution. When cars arrive in a time slot that the traffic light is red, they enter a queue. When the light becomes green, exactly one car leaves the queue. In the case that cars arrive when the light is green and the queue is empty, they pass the light without queueing.

The parameters in this model are:

- the arrival rate λ;
- the number of successive time slots the light is red r;
- and the number of successive time slots the light is green g.

The successive time slots in which the light is green and red is called the green period and red period, respectively. In order for the system to be stable the number of arriving cars must be less than the number of cars that can pass the light. Therefore, it should hold that $(g+r)\lambda < g$.

The performance measures of interest are:

- the mean queue length at the end of green periods;
- and the mean queue length at the end of red periods.

A closed form formula for the mean queue length at the end of green periods, denoted by $\mathbb{E}[X_g]$, is described by Van Leeuwaarden [4]. Computing this value requires solving a set of linear equations. In this paper also a table is provided with results of specific values for λ, r and g. We can use the values in this table to verify the results of the simulation. The values in which we are interested can be found in Table 3.1. The values for r and g are both 5.

λ	$\mathbb{E}[X_g]$
0.30	0.1800
0.40	1.0971
0.45	3.3998
0.49	23.2249

Table 3.1: Mean queue length at the end of green periods

The mean queue length at the end of red periods is simply the mean queue length at the end of green periods plus the number of red slots times the arrival rate:

$$\mathbb{E}[X_0] = \mathbb{E}[X_g] + r\lambda. \tag{3.5}$$

Chapter 4

Implementation without the use of frameworks

In this chapter we describe the implementations of the models explained in the previous chapter without the use of frameworks. We already discussed the mathematical model with its notation and assumptions and we already defined the performance measures in which we are interested. Before we start writing a simulation, we identify the following parts of the model:

- the entities in the model;
- the attributes of the entities;
- and the events in the model.

In the following sections we start by identifying the above parts. Next, we describe the simulation program step by step and explain the implementation details. Only the core function of the simulation is presented in the sections. In all models we need to sample from the exponential distribution. To obtain a sample from this distribution we applied the inverse transformation method.

4.1 M/M/c tandem queue

In the first model the entities that play a role are the server, the customer and the queue. For the customer we need to keep track of the arrival time and service time at each station. Since we are also interested in the sojourn time, we also need to save the time the customer enters the system. Therefore, we create a `Customer` class with the attributes `arrivalTime`, `serviceTime` and `systemIn`. A server class does not have to be implemented, because the arrival and departure times of a customer are attributes of the customer. For each queue we need to know the length of it and which customers are currently in the queue. Therefore, we create a `Queue` class that contains a list of the customers that are currently in the queue. We create one customer object for each customer that enters the system and one queue object for each station.

The following events change the state of the system:

- a customer arrives at a queue;
- a server takes a customer into service;
- a server is finished serving a customer;
- and a customer leaves a station.

Some of these event take place simultaneously. When a customer arrives at an empty queue, the server takes the customer into service immediately. In that case the first two events coincide. The third and fourth events always coincide, because a customer leaves the station at the same moment the server has finished the service. Also when a server finished serving a customer and there are customers waiting, a new customer will be taken into service at the same moment. With these co-occurring events we do not have to implement all of them, but only the arrival and departure events of a customer. We create the class `Event` with attributes `type`, `station`, `time` and `customer`. In order to control the events we schedule the events in a Future Event Set, denoted by FES. The class `FES` contains a list with scheduled events sorted on time.

Listing 4.1 shows the structure of the core method for the simulation. Before the while-loop is entered, the objects that are needed for the simulation are initialized and the first customer arrival is scheduled. The while-loop runs until the time of a scheduled event exceeds the specified simulation time given by `simulationTime`. The FES is named `eventq` and lists the scheduled events. The method `checkEvent()` returns the first event in this list and with the method `getTime()` we obtain the time for which this event is scheduled. When the time of the next event exceeds the simulation time we return the results. In each iteration one event is handled. The event to be handled is returned and removed from the FES with the use of the method `nextEvent()`. We register information about the current event in the object `results` with the use of `registerEvent()`. In the results object the performance measures are computed. The main part of the simulation function is to handle the events. Both types of events, which are a customer arrival and a customer departure, causes a different state change in the system and therefore need to be handled differently.

The code for handling an arrival event is shown in Listing 4.2. When the event is an arrival, the following actions need to be taken:

- the arriving customer is added to the queue of the corresponding station;
- if at least one of the servers is available, the customer is taken into service immediately;
- and if the customer arrives at the first station, a new customer arrival is scheduled.

A customer is always added to the corresponding queue, even in the case a server is available. The moment that a customer is taken into service the departure event is scheduled. The departure is scheduled at the time of the arrival plus the service time. Only in the case that the arrival is at the first station a new customer arrival is scheduled. New customers do not have to be created at the other stations, since customers are already present in the system.

Listing 4.1: The simulation function

```
public Results simulate(double simulationTime) {
    ...

    while (eventq.checkEvent().getTime() < simulationTime) {
        Event e = eventq.nextEvent();
        int s = e.getStation();
        double t = e.getTime();
        Customer c = e.getCustomer();

        results.registerEvent(e);

        // Handle the event.
        if (e.getType() == Event.ARRIVAL) { // ARRIVAL
            ...
        } else { // DEPARTURE
            ....
        }
    }

    return results;
}
```

Listing 4.2: Handling of an arrival event

```
// Add customer to the queue.
queues[s].addCustomer(c);

// If a server is available, take the customer into service.
if (queues[s].getSize() <= nrOfServers) {
    // Schedule departure event of the customer.
    time = t + c.getServiceTime();
    newEvent = new Event(Event.DEPARTURE, s, time, c);
    eventq.addEvent(newEvent);
}

// If the arrival is at the first station, create and schedule a new customer.
if (s == 0) {
    // Create new customer.
    arrivalTime = t + arrivalDist.nextRandom();
    serviceTime = serviceDist.nextRandom();
    systemIn = arrivalTime;
    newCust = new Customer(arrivalTime, serviceTime, systemIn);

    // Schedule arrival event.
    station = 0;
    time = newCust.getArrivalTime();
    newEvent = new Event(Event.ARRIVAL, station, time, newCust);
    eventq.addEvent(newEvent);
}
```

The code for handling a departure event is shown in Listing 4.3. When the event is a departure, the following actions need to be taken:

- the departing customer is removed from the queue of the corresponding station;
- the server that becomes available serves the next waiting customer if there is any;
- and the departing customer enters the queue of the next station or leaves the system.

At the moment a customer is served he is removed from the queue. After the departure of a customer it is checked whether there are customers waiting in the queue. The number of customers in the queue is compared with the number of servers at the station. Since customers remain in the queue while they are being served, a customer is waiting when the queue size is greater or equal than the number of servers. When the departing customer remains in the system by entering the next queue, the arrival time and service time of the customer are updated. We assume that there is no travel time between a station and the next queue, so the current departure time becomes the next arrival time. Since the arrival time is updated at each station, the arrival time at the first station need to be saved in order to compute the sojourn time.

Listing 4.3: Handling of an departure event

```
// Remove customer from the queue.
queues[s].removeCustomer(c);

// If there is a customer waiting, take the customer into service.
if (queues[s].getSize() >= nrOfServers) {
    // Schedule departure event of the customer.
    nextCust = queues[s].getCustomerAtPosition(nrOfServers - 1);
    time = t + nextCust.getServiceTime();
    newEvent = new Event(Event.DEPARTURE, s, time, nextCust);
    eventq.addEvent(newEvent);
}

// If the departure is not at the last station, schedule next arrival event.
if (s < nrOfStations - 1) {
    // Update the arrival time and service time.
    c.arrivalTime = t;
    c.serviceTime = serviceDist.nextRandom();

    // Schedule arrival event.
    station = s + 1;
    time = c.getArrivalTime();
    newEvent = new Event(Event.ARRIVAL, station, time, c);
    eventq.addEvent(newEvent);
}
```

4.2 Polling system with gated service

In the second model the entities that play a role are again the server, the customer and the queue. For the customer we need to keep track of the arrival time and service time. Therefore, we created a `Customer` class with the attributes `arrivalTime` and `seviceTime`. In this model we also had to implement a class for the server. Since we are interested in the cycle time of the server, we need to keep track of the start time and end time of serving one round of stations. At each station the server only serves the customers that are in the queue at the moment the server arrived. Therefore, the server needs to keep track of the number of customers he has already served. We created a `Server` class with the attributes `startCycleTime`, `endCycleTime` and `serveCustomers`. The `Queue` class could be reused from the previous model. We create one server object, one customer object for each customer that enters the system and one queue object for each station.

The following events change the state of the system:

- a customer arrives at a queue;
- the server takes a customer into service;
- the server is finished serving a customer;
- a customer leaves a station;
- the server leaves a station;
- and the server arrives at a station.

Some of these event take place at the same time. A customer arrival never coincide with one of the other events at the same station. When a server arrives at a station the event coincide with the event that the server takes the customer into service. This happens when the queue of the corresponding station is not empty. The event that the server is finished serving a customer always coincide with a customer leaving a station. When the server has served all customers it also coincide with the event that the server leaves the station and otherwise that the next customer is taken into service. With these co-occurring events we only had to implement the customer arrival, customer departure and server arrival. We could reuse the class `Event` from the previous model with the attributes `type`, `station`, `time` and `customer`. The difference with the previous model is that an event not always corresponds to a customer. So, in case the type of the event is a server arrival the customer attribute is `null`. The events are again scheduled in a `FES` object.

Listing 4.4 shows the structure of the core method for the simulation. The structure is the same as in the previous model. The simulation function handles next to the customer arrival and customer departure also the server arrival.

Listing 4.4: The simulation function

```
public Results simulate(double simulationTime) {
    ...

    while (eventq.checkEvent().getTime() < simulationTime) {
        Event e = eventq.nextEvent();
        int s = e.getStation();
        double t = e.getTime();
        Customer c = e.getCustomer();

        results.registerEvent(e);

        // Handle the event.
        if(e.getType() == Event.ARRIVAL) { // ARRIVAL
            ...
        } else if (e.getType() == Event.DEPARTURE) { // DEPARTURE
            ...
        } else { // SERVER ARRIVAL
            ...
        }
    }

    return results;
}
```

The code for handling an arrival event is shown in Listing 4.5. When the event is an arrival, the following actions need to be taken:

- the arriving customer is added to the queue of the corresponding station;
- and a new customer arrival is scheduled.

In this model it never happens that an arriving customer is immediately taken into service, because the server only serves the customers that were in the queue at the moment the server arrived. A new customer arrival is always scheduled at the same station as the current arriving customer.

Listing 4.5: Handling of an arrival event

```
// Add customer to the queue.
queues[s].addCustomer(c);

// Create new customer.
arrivalTime = t + arrivalDist.nextRandom();
serviceTime = serviceDist.nextRandom();
newCust = new Customer(arrivalTime, serviceTime);

// Schedule arrival event.
time = newCust.getArrivalTime();
newEvent = new Event(Event.ARRIVAL, s, time, newCust);
eventq.addEvent(newEvent);
```

The code for handling a departure event is shown in Listing 4.6. When the event is a departure, the following actions need to be taken:

- the departing customer is removed from the queue;
- if the server has served all customers, the server switches to the next station;
- and if the server has not served all customers, the next customer is taken into service.

The method `getServeCustomers()` returns the number of customers that the server still needs to serve. This value is updated each time a customer is served. When this value is equal to zero, the server has to switch to the next station. The time it takes to move to the next station is given by `switchTime`. For the switch of the server an event is scheduled of type `ServerARRIVAL` at the current time plus the switch time.

Listing 4.6: Handling of an departure event

```
// Remove customer from the queue.
queues[s].removeCustomer(c);

// If the server has served all customers, switch the server.
if (server.getServeCustomers() == 0) {
    nextStation = (s + 1) % nrOfStations;
    time = t + switchTime;
    newEvent = new Event(Event.ServerARRIVAL, nextStation, time, null);
    eventq.addEvent(newEvent);
} else {
    // Schedule departure event of the customer.
    nextCust = queues[s].getFirstCustomer();
    time = t + nextCust.getServiceTime();
    newEvent = new Event(Event.DEPARTURE, s, time, nextCust);
    eventq.addEvent(newEvent);

    // Decrease the customers to be served with one.
    server.serveCustomers--;
}
```

The code for handling a server arrival is shown in Listing 4.7. When the event is a server arrival, the following actions need to be taken:

- if the server is back at the first station, the cycle time is registered;
- if there are customers waiting in the queue, the first customer is taken into service;
- and if the queue is empty, the server switches to the next station.

If the server is back at the first station, the information about the cycle time is registered in the object `results` with the use of the method `registerCycleTime()`. The time the previous cycle ended is the start time of the current cycle. The moment the server arrives at a station, the number of customer in the corresponding queue needs to be counted. In the situation there is at least one customer waiting the first customer is taken into service. Otherwise, when the value `serveCustomers` is zero, the server moves to the next station.

Listing 4.7: Handling of a server arrival

```
// If the server is back at begin, register cycle time.
if (s == 0) {
    server.endCycleTime = t;
    results.registerCycleTime(server);
    server.startCycleTime = t;
}

// Count the number of customers to be served.
server.serveCustomers = queues[s].getSize();

// If there is a customer waiting, take the customer into service.
if (server.getServeCustomers() > 0) {
    // Schedule departure event of the customer.
    nextCust = queues[s].getFirstCustomer();
    time = t + nextCust.getServiceTime();
    newEvent = new Event(Event.DEPARTURE, s, time, nextCust);
    eventq.addEvent(newEvent);

    // Decrease the number of customers to be served with one.
    server.serveCustomers--;
} else { // Switch the server again.
    nextStation = (s + 1) % nrOfStations;
    time = t + switchTime;
    newEvent = new Event(Event.ServerARRIVAL, nextStation, time, null);
    eventq.addEvent(newEvent);
}
```

4.3 Discrete-time fixed-cycle traffic light

In the last model the entities that play a role are the car, the traffic light and the queue. The car does not contain any attributes, since we are not interested in the time the car spends in the system. However, we implemented a `Car` class without any attributes so that we can easily extend the model when needed. We created a `Queue` class that contains a list of the cars that are currently in the queue. When we would not have chosen to implement the cars, we could have only created a counter that keeps track of the queue length. For the traffic light we need to keep track of the current status of the light and how many slots until the status has to change. Therefore, we created a `Light` class with the attributes `status` and `slots`. We create one car object for each arriving car, one light object and one queue object.

The following events change the state of the system:

- a car arrives at the queue;
- a car passes the traffic light;
- and the light changes its color.

The time in this model is discretized. It depends on the time slots when the light changes its colour. The arrival of cars happens at any moment in time and does not depend on the time slots. How the arrival of a car is handled depends on the current status of the light. Also the event that a car passes the traffic light depends on the colour of the light. Because of the

dependence on the light, the events are handled differently than in the previous model. We still need the class `Event`, but with only the attributes `time` and `car`, and the class `FES` to schedule the arrival events.

Listing 4.8 shows the structure of the core method for the simulation. Before we enter the while-loop, the objects that are needed for the simulation are initialized and the first car arrivals are scheduled. The while-loop runs for the specified number of time slots given by `timeslots`. In each iteration the events that occur during one time slot are handled. How the events are handled depends on whether the light is green or red. The variable `n` is an integer that indicates the current time slot and is increased in each iteration. The method `checkEvent()` only returns the next scheduled event, but does not remove it yet from the FES. At the moment an event is handled it is removed. After all events in the current time slot are handled, new cars are generated and there need to be checked whether the traffic light has to change its status.

Listing 4.8: The simulation function

```
public Results simulate(int timeslots) {
    ...

    while (n < timeslots) {
        Event e = eventq.checkEvent();
        double t = e.getTime();
        Car c = e.getCar();

        // Handle the events in the current time slot.
        if (light.getStatus() == Light.GREEN) { // GREEN
            ...
        } else { // RED
            ...
        }

        // Generate next arrivals.
        ...
        // Check whether the light has to change its status.
        ...
    }

    return results;
}
```

The code for handling the arrivals when the light is green is shown in Listing 4.9. When the status of the light is green, the following actions need to be taken:

- if the queue is empty, arriving cars pass the light without delay;
- if the queue is not empty, arriving cars are added to the queue;
- and the first car in the queue leaves.

The current time slot is indicated with the time between `n` and `n + 1`. So, when the time of an arrival is below `n + 1` it occurs during the current time slot. If the queue is not empty the car is added to the queue and otherwise it passes without being added to the queue. In both cases the arrival event is handled and it can be removed from the list with the use of

the method `removeEvent()`. It is checked whether the next arrival occurs in the current time slot, otherwise the while-loop is exited. When all arrival events are handled, we remove the first car from the queue. We could not do this before the arrivals are handled, because the queue could be falsely interpreted as empty.

Listing 4.9: Handling of events when the light is green

```
// Handle the arrivals that are scheduled in the current time slot.
while (t < n + 1) {
    // If the queue is not empty, the car enters the queue.
    if (q.getSize() > 0) {
        q.addCar(c);
        eventq.removeEvent();
    } else {
        eventq.removeEvent();
    }

    // Check next arrival.
    e = eventq.checkEvent();
    t = e.getTime();
    c = e.getCar();
}

// If there are cars waiting, the first car leaves the queue.
if (q.getSize() > 0) {
    q.removeFirstCar();
}
```

The code for handling the arrivals when the light is red is shown in Listing 4.10. When the status of the light is red, arriving cars only have to be added to the queue.

Listing 4.10: Handling of events when the light is red

```
// Handle the arrivals that are scheduled in the current time slot.
while (t < n + 1) {
    q.addCar(c);
    eventq.removeEvent();

    // Check next arrival.
    e = eventq.checkEvent();
    t = e.getTime();
    c = e.getCar();
}
```

The code for preparing the next time slot is shown in Listing 4.11. We need to generate new arrivals and check whether the light has to change its status. The current iteration represents the time slot between time `n` and `n + 1`. We want to generate arrivals for the time slot between `n + 1` and `n + 2`. The value of `arrivalTime` is the time of the first event that did not happen in the current time slot anymore. The arrival time is updated with `arrivalDist.nextRandom()` after each scheduled arrival. We generate car arrivals until we generate an arrival that occurs not in the next slot anymore. After we generated the next arrivals we increased the counter for the current slot and decreased the slots until the status of the light has to change with one. In case the `slots` has the value zero, the light has to change its color. At this moment we register the length of the queue with the use of the method `registerLength()`. We change the status and update the number of slots that indicate when the light has to change its status again.

Listing 4.11: Preparation for the next time slot

```
// Generate arrivals in the next time slot.
arrivalTime = t;
while (arrivalTime < n + 2) {
    arrivalTime += arrivalDist.nextRandom();
    time = arrivalTime;
    newEvent = new Event(time, new Car());
    eventq.addEvent(newEvent);
}

n++;
light.slots--;

// Check whether the light has to change its status.
if (light.getSlots() == 0) {
    // Register the queue length.
    results.registerLength(q, light);

    // Change the light to the other color.
    if (light.getStatus() == Light.GREEN) { // Change to red.
        light.status = Light.RED;
        light.slots = nrOfRedSlots;
    } else { // Change to green.
        light.status = Light.GREEN;
        light.slots = nrOfGreenSlots;
    }
}
```

4.4 Improvements

The performance of simulations becomes important when they have to run for a long simulated time or when multiple replications are needed. The performance can be increased by using a faster random number generator and by using the best suitable data structures. In this section we investigate to what extent the performance can be improved. In order to measure the results of a performance improvement we need to simulate a large number of events. We do this by running a simulation of an M/M/c tandem queue model consisting of three stations on which 100 servers are working. All servers work at a service rate of 1.0 and customers arrive with a rate of 99.0. The system satisfies the stability condition, but a large number of events will be scheduled. We use the implementation explained in the previous section as a baseline.

For the random number generator we originally used the standard random number generator in Java from the class `Random`. This is however not the fastest implementation for generating random numbers. One of the best implementations at this moment with respect to performance is the Mersenne Twister. This is the standard random number generator used in the statistical software R. We used the implementation from the Apache Commons, which is described in the paper by M. Matsumoto and T. Nishimura [5].

Listing 4.12: Adding a new event to the list

```
public void addEvent(Event newEvent) {
    int index = 0;

    while (index < events.size()) {
        Event e = events.get(index);

        if (e.getTime() <= newEvent.getTime()) {
            index++;
        } else {
            break;
        }
    }

    events.add(index, newEvent);
}
```

For the implementation of the queues and the FES we originally used an `ArrayList`. The list of the FES is sorted, which is obtained in a naive way by inserting a new scheduled event at the right place in the list. Listing 4.12 shows the code for adding a new event to this list. For both the list of the queue and the list of the FES the operations `add()`, `get()` and `remove()` are used. Instead of using the `ArrayList` we could also have used the `LinkedList`. However, the implementation of this data structure would decrease the performance. The `LinkedList` has advantage over the `ArrayList` in the situation that elements are added and deleted in the front of the list. This does not outweigh the disadvantage of the often used operation `get()`, which is much slower in case of the `LinkedList`. For the implementation of the queue, we believe that the `ArrayList` is the best suitable data structure. For the implementation of the FES there exists a better data structure. The implementation of adding an event to the FES is not very efficient. A more efficient implementation would be to make use of a `PriorityQueue`. This data structure ensures that the list remains sorted on time.

In order to measure the performance improvement we implemented the simulation with the use of the alternative random number generator and data structure for the FES. In Table 4.1 is shown which data structure and generator is used in the implementations. Implementation 1 is the original, which is used in the previous section. In implementations 2 and 3 the Mersenne Twister and the `PriorityQueue` are used, respectively. In implementation 4 we combined the two alternatives in order to measure the total performance improvement.

Implementation	**FES**	**Random number generator**
1	`ArrayList`	standard
2	`ArrayList`	Mersenne Twister
3	`PriorityQueue`	standard
4	`PriorityQueue`	Mersenne Twister

Table 4.1: Details of the implementations

We ran the simulation for 100.000 time units. Because of the high parameter values, a large number of arrivals and departures is scheduled. Therefore, the improvement when using an alternative implementation can be measured well. Figure 4.1 shows the performance

improvement for the alternative implementations. The results are based on 10 replications for each implementation. The individual observations can be found in Appendix A. When the original implementation is used the running time is 36.0 seconds. Using the Mersenne Twister instead of the standard random number generator in Java results in a decrease of the running time to 34.3 seconds, which is a decrease of about 5%. Using a `PriorityQueue` instead of using an `ArrayList` for the FES results in a decrease of the running time to 29.8 seconds, which is a decrease of about 17%. Combining the two alternatives results in a total improvement of 21% to a running time of 28.4 seconds.

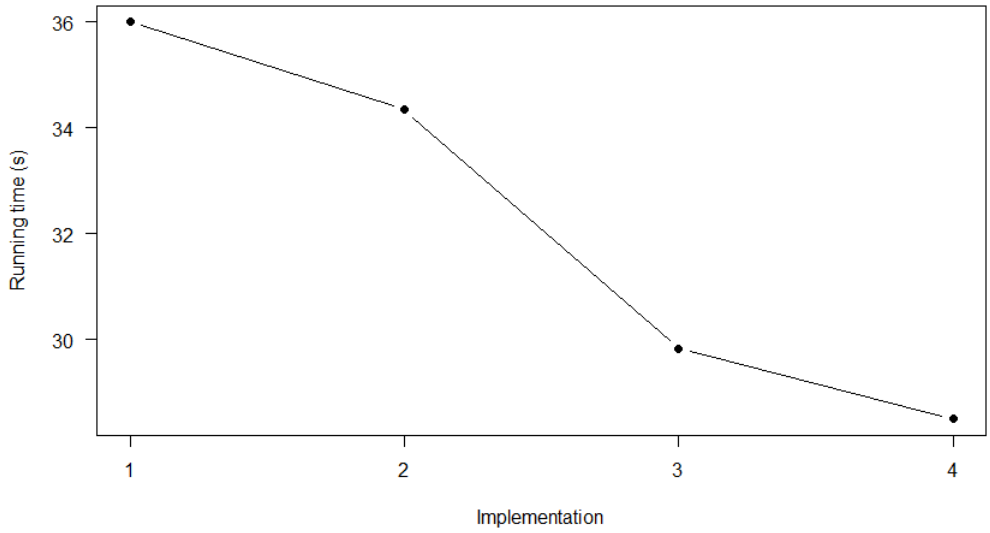

Figure 4.1: Running times of the implementations

In the situation that a sorted list is needed in the simulation it is preferred to use a `PriorityQueue` instead of a sorted `ArrayList`. Without much implementation effort there is a significant increase in performance. Also with the use of a better random number generator the performance improves, although the improvement is not as high as for the alternative data structure.

Chapter 5

Choice of frameworks

In the previous chapter, we implemented the models without the use of frameworks. For these implementations we made use of the event-scheduling approach. Using the process-interaction approach would decrease the implementation effort and improve the understandability of the program. The means for that implementation approach are provided by frameworks. Since numerous frameworks are developed, we needed to make a selection of which we would use in this project. Therefore, we defined a few criteria that the framework needs to satisfy. In this chapter we select the frameworks that seem to offer the best functionality and explain their characteristics.

5.1 Selection

We selected the frameworks based on the following criteria:

- flexibility;
- recent release or update;
- and support for the process-interaction approach.

The first criterion is flexibility, by which we mean that it should be possible to implement all types of discrete event models. We are not able to check whether a framework satisfies this criterion. Therefore, we consider a framework flexible when we are able to implement each of the three models with the use of the specific classes of the framework.

In the scope of this project, we do not consider frameworks that are currently not used or developed anymore. While studying the literature, numerous frameworks came across. Most papers that are written about frameworks date from the period around the year 2000. Some of these frameworks that are described in this period are recently updated or used, but others were discontinued (possible due to better alternatives or not enough support). For example the framework named Silk described by Andradóttir and Healy [6] is such a framework that was discontinued. It is developed by the company ThreadTec, which does not exist anymore. The first papers written about Silk date from 1997 and it is mentioned in many other papers written around that time. The development of Silk stopped within a few years.

We want to use frameworks to support the implementation of the process-interaction approach. Most frameworks offer this approach, but also frameworks exist that only support the implementation of the event-scheduling approach. The framework SimKit described by Buss and Stork [7] is such a framework that does not support the process-interaction approach. It is debatable whether the framework DSOL described by Jacobs et al [8] should be selected based on the three criteria. DSOL is currently still developed at the university of Delft and seems to offer enough flexibility. Both approaches are supported, but the focus of this framework is the support of the event-scheduling approach. The means for implementing the process-interaction are provided, but at runtime this implementation is handled as an event-based implementation. Instead of handling the processes as threads and synchronizing the threads, the behaviour of the processes is translated to a FES. The translation from processes to events has a significant negative impact on the performance of the simulation. The framework SSJ also offers the possibility to use this implementation next to the thread based implementation. In the SSJ User's Guide about the package `simprocs` states that the implementation of the process-interaction by DSOL is about 50 times slower than the thread based implementation. Therefore, other frameworks that do not have this large decrease in performance are preferred over DSOL. Also because of the lack of a good tutorial for using the process-interaction approach, we decided to not select this framework.

Based on the three criteria we selected the frameworks DESMO-J and SSJ. Both frameworks have recent releases and are currently still being developed. In Chapter 6 it can be seen that these frameworks offer enough possibilities to implement the three models. Both the event-scheduling and process-interaction approaches are supported in these frameworks, but we only used the process-interaction approach. In the next two sections we explain the characteristics of both frameworks.

5.2 Description of DESMO-J

DESMO-J stands for Discrete Event Simulation and Modeling in Java. It is developed at the university of Hamburg. The first version dates from 1999 and it is currently still under development. Approximately once a year a new version is released. In this project we used the latest version (2.5.1c) which dates from November 2015 and can be downloaded from the website of Sourceforge [9]. In the paper by Göbel et al. [10] the core functionality is described, which we summarize in this section.

Several classes are offered that are ready to use and of which the code should not be touched by the user. One of these classes is the `Experiment`, which is responsible for running the simulation. Also classes for queues, random number generators and collection of statistical data are provided. For each process that play a role in the model a class should be created that extends the class `SimProcess`. Within this class the behaviour of the process during the simulation is described in the method `lifeCycle()`. A thread runs for each active process and ends when the process ends.

DESMO-J provides methods such that these threads are synchronized. Either a process is held for a specified time or it is passivated until an other process activates it again. A process can only hold or passivate itself by calling the method `hold()` or `passivate()`, re-

spectively. Another process can activate a passive process by calling the method `activate()`. The interaction between processes is possible with the use of queues. DESMO-J provides a class specifically for process queues in which processes can be inserted and removed either by itself or by another process. Information about the queue, such as values as the mean queue length and the mean waiting is automatically collected.

Two types of random number generators are provided by the framework: a Linear Congruential Generator and a Mersenne Twister. When no generator is set the Linear Congruential Generator is used. In the implementation of the models we used this default generator. The initial seed is always 42, but an other value can be set by calling the method `setSeed()`.

An HTML report about the simulation will be created when the method `report()` is called. This report provides information about the simulation. Next to the statistical data collection from the process queues, additional so called "tallies" can be created. During the simulation values are added to the tally of which it computes statistical information. Both the results of the data collection from the process queues and the tallies are included in the report. Next to the possibility to create a report, it is possible to create a trace of the simulation. Within such a trace all steps of the simulation can be included for a specified time period. Creating the trace is a helpful tool in debugging the code.

5.3 Description of SSJ

SSJ is the abbreviation of Stochastic Simulation in Java. It is developed at the university of Montreal. The first version came out in 2000 and the framework is currently still developed. The latest version is 3.2.0 and is last updated in September 2016. In this last version the classes needed for the process-interaction approach are not included. However, these classes are mentioned in the documentation of the latest version. Therefore we used the previous version 2.6.2 from a year before, which can be downloaded from the website of SSJ [11]. Some of the functionality provided for the process-interaction approach is described by L'Ecuyer et al. [12]. We will summarize this description and explain how we handle the processes interaction within this framework.

The implementation with a process-interaction approach can be combined with that of an event-scheduling approach. The packages `simprocs` and `simevents` are provided for the process-interaction and event-scheduling approach, respectively. We only make use of the event-scheduling approach to schedule the "end of simulation" event, which will stop the simulation. For each process that play a role in the model a class should be created that extends the class `SimProcess`. Within this class the behaviour of the process during the simulation is described in the method `actions()`. A thread runs for each active process and ends when the process ends. Describing the behaviors of processes is similar as that in DESMO-J, but the means for synchronization differ. The synchronization and interaction between processes coincide in this framework. For this, we make use of two classes: `Resource` and `Condition`.

A resource has a specified capacity and a process can request part of that capacity. In the example of an M/M/c queueing model a resource would be the station with capacity c and each arriving customer would request one unit of that capacity. Requests for a resource are

handled until there is no capacity left. When the request of a process can not be handled, the process is automatically put into a queue. Eventually a process that holds part of the capacity of a resource releases it again. At the moment enough capacity is available for the first waiting process, the process can proceed. By calling the method `request()` a resource is requested and by calling the method `release()` the capacity of the resource is made available again. The number of units that is requested or released is specified in the parameter of that method.

The other class that is used for synchronization and interaction is `Condition`. A condition can either be set to true or false. When a process encounters a condition that is false it has to wait for the condition to become true. While the process waits, it is automatically put into a list. In contrast to the queue for a resource, the list of processes that wait for a condition is not ordered. When the condition is set to true, all waiting processes proceed. Information about the waiting lists for the resources and conditions is automatically collected.

SSJ provides several implementations for the random number generation. In the implementation we used the generator called MRG32k3a, since this is also the one used in the tutorial examples. According to the documentation about the generators, this generator is also the most extensively tested and is proposed by one of the developers of the framework [13]. At the moment faster generators are also available such as variants of the Mersenne Twister. The MRG32k3a is a Combined Multiple Recursive Generator. The seed consists of six `long` numbers and can be set by the method `setPackageSeed()`. There are some constraints about the values in the seed. The first three values must be less than 4294967087 and not all 0 and the last three values must be less than 4294944443 and not all 0. When the seed is not explicitly set the default seed is used, which consists of six times the value 12345.

In SSJ there is no function to create a report or trace such as in DESMO-J. However, printed output with statistical data about the waiting lists for the resources and conditions and statistical data about the tallies can be generated.

Chapter 6

Implementation with the use of frameworks

In this chapter we explain the implementation details of the three models with the use of the selected frameworks DESMO-J and SSJ. First, we discuss the implementation with DESMO-J and next the implementation with SSJ. We do not first explain the behaviour of the processes that play a role in the model as what we did for the entities and events in Chapter 4. Before we implemented the models, we did this first step. However, it should be clear when we explain the behaviour of the processes by means of the process implementation.

6.1 DESMO-J

In the previous chapter we mentioned some of the characteristics of the framework DESMO-J about how the behaviour of the processes is described and how the active processes interact. The initialization of the simulation and the generation of processes that enter the system is the same in all models besides the names and parameter values. Therefore, we discuss this part of the programs only for the first model.

6.1.1 M/M/c tandem queue

The initialization of the simulation of the first model is shown in Listing 6.1. In the class `TandemQueue` we initialize the objects we need in the simulation and specify the distributions and parameters. For all three models we need the exponential distribution. An important note is that the parameter of the exponential distribution is not λ but λ^{-1}. We need to override in this class three abstract methods of the class `Model`: `description()`, `doInitialSchedules()` and `init()`. Within the method `description()` a textual description needs to be given about the model that will appear in the report. The methods `doInitialSchedules()` and `init()` create the dynamic and static model components, respectively. The dynamic components that are activated are the server processes and the customer generator process. For each server in the system, a server process is created. For each of the stations in the system, a `ProcessQueue` is created in which the servers of the corresponding station are inserted. The customer generator process will generate the arriving customers in the system and will activate the customer processes.

Listing 6.1: Initialization of the simulation

```
public class TandemQueue extends Model {
    ...

    // Returns a description of the model to be used in the report.
    public String description() {
        ...
    }

    // Activates dynamic model components (simulation processes).
    public void doInitialSchedules() {
        ...
    }

    // Initializes static model components.
    public void init() {
        ...
    }

    public static void main(java.lang.String[] args) {
        // Create model and experiment.
        TandemQueue model = new TandemQueue(null, "Tandem queue", true, true);
        Experiment exp = new Experiment("Tandem queue");

        // Connect both.
        model.connectToExperiment(exp);
        // Stop condition.
        exp.stop(new TimeInstant(100000));
        // Start the experiment at simulation time 0.0.
        exp.start();
        // Generate the report.
        exp.report();
        // Stop all threads still alive and close all output files.
        exp.finish();
    }
}
```

In the main method we define the model `TandemQueue` and create an experiment. The model has four parameters. The first parameter is set to `null`, which indicates there is no main model associated with this model. The second parameter is the name of the model. The third and fourth parameter indicate whether this model should appear in the report and in the trace, respectively. We can specify the simulation time of the experiment by setting the stop condition. This simulation runs for a 100.000 time units.

The implementation of the customer generator process is shown in Listing 6.2. There is one customer generator process that never stops during the simulation. It creates a new customer and activates the customer process immediately after the creation. The parameters of the customer specify the model it belongs to, the name of the process and whether is should appear in the trace. The generator process is set to hold until the next customer should arrive.

Listing 6.2: Customer generator process

```
while (true) {
    Customer customer = new Customer(model, "customer", true);
    customer.activateAfter(this);

    // Wait until the customer arrives to schedule a new arrival.
    hold(new TimeSpan(model.getArrivalTime()));
}
```

The implementation of the customer process is shown in Listing 6.3. A customer enters the system at the first station and leaves the system after it went through all stations. We get the arrival time of the customer and specify that the customer starts at the first station. The number of stations in this model is given by the value `nrOfStations`. While the customer is not at the last station it enters the customer queue of the next station. When the queue of idle servers is empty, which means that all servers are busy serving another customer, the customer process passivates. The customer process waits until it will be activated by a server process. When the customer is activated again, it means that the customer is served and he can go to the next station. In the case that there is a server in the idle server queue, the server is removed from the queue and immediately activated by the customer. When the customer leaves the last station the tally that keeps track of the mean sojourn time is updated.

Listing 6.3: Customer process

```
// Specify the arrival conditions.
double arrivalTime = presentTime().getTimeAsDouble();
int station = 0;

// Customer goes through all stations.
while (station < model.nrOfStations) {
    // Insert the customer in the queue of the current station.
    model.customerQueues[station].insert(this);

    // Check if a server is available.
    if (!model.idleServerQueues[station].isEmpty()) {
        // Activate an idle server.
        Server server = model.idleServerQueues[station].first();
        model.idleServerQueues[station].remove(server);
        server.activateAfter(this);
    }

    // Wait for service.
    passivate();

    // The customer is served at the station and goes to the next.
    station++;
}

// Register the system time of the customer.
model.sojournTime.update(presentTime().getTimeAsDouble() - arrivalTime);
```

The implementation of the server process is shown in Listing 6.4. In contrast to a customer process, server processes do not end during the simulation. Therefore, the behaviour is described in a while-loop of which the condition is always true. A server can either serve a customer or be idle in the idle server queue. When the customer queue at the station of the server is empty, the server will insert itself in the idle server queue. The process passivates

until a customer arrives and activates the server again. When the customer queue is not empty, the server process removes the first customer from the queue and the process holds for a time given by `getServiceTime()`. When this time is over the server is finished serving the customer and activates the customer to finish its life cycle.

Listing 6.4: Server process

```
while (true) {
    // Check if there is a customer waiting.
    if (model.customerQueues[station].isEmpty()) { // EMPTY
        // Insert server into the idle server queue.
        model.idleServerQueues[station].insert(this);

        // Wait until a customer arrives.
        passivate();
    } else { // NOT EMPTY
        // Remove the first customer from the queue.
        Customer nextCustomer = model.customerQueues[station].first();
        model.customerQueues[station].remove(nextCustomer);

        // Serve the customer.
        hold(new TimeSpan(model.getServiceTime()));

        // Reactivate the customer for finishing his life cycle.
        nextCustomer.activate();
    }
}
```

6.1.2 Polling system with gated service

As already mentioned we do not explain the implementation of the initialization and the customer generator process for this model. The difference is that we have a customer generator process for multiple queues instead of for a single one. The implementation of the customer process is shown in Listing 6.5. The only behaviour a customer has is arriving at a queue and waiting for service. Within DESMO-J this is very easy to implement. The customer only inserts itself in the queue and waits for service. When the server activates the customer, it means that the customer is served and the customer process ends.

Listing 6.5: Customer process

```
// Insert the customer in the queue.
myModel.customerQueues[station].insert(this);

// The customer waits for service.
passivate();
```

The implementation of the server process is shown in Listing 6.4. There is one server that never stops working. Before the server starts serving customers, it is specified that the server starts at station 0. When the server arrives at a station, the number of customers in the corresponding queue are counted. All customers that are at that moment in the queue will be served that round. The service of the customer is described in the same way as in the previous model. When all customers are served or when no customers had to be served the server moves to the next station. The server is set to hold until the switch-over time has

passed. The position of the server is updated and in case that the server is back at the first station the tally that keeps track of the mean cycle time is updated.

Listing 6.6: Server process

```
// Specify the start conditions of the server
int station = 0;
double startTime = 0.0;

while (true) {
    // Count the number of customers to be served.
    int serveCustomers = model.customerQueues[station].size();

    // If there are customers waiting, serve the customers.
    if (serveCustomers > 0) {
        for (int i = 0; i < serveCustomers; i++) {
            // Get the first customer
            Customer nextCustomer = model.customerQueues[station].first();
            model.customerQueues[station].remove(nextCustomer);

            // Serve the customer.
            hold(new TimeSpan(model.getServiceTime()));

            // Reactivate the customer for finishing his life cycle.
            nextCustomer.activate();
        }
    }

    // Switch the server.
    hold(new TimeSpan(model.switchTime));
    station = (station + 1) % model.nrOfStations;

    // If the server is back at begin, register the cycle time.
    if (station == 0) {
        model.cycleTime.update(presentTime().getTimeAsDouble() - startTime);
        startTime = presentTime().getTimeAsDouble();
    }
}
```

6.1.3 Discrete-time fixed-cycle traffic light

In the last model we have to describe the behaviour of a car and a traffic light process. The customer generator process from the first model is reused, with the difference that it creates cars instead of customers. The implementation of the car process is shown in Listing 6.7. The value of `status` can either be red or green, which indicates the colour of the traffic light. When a car arrives and the light is red, it inserts itself in the queue and waits. In the case the light is green it depends on the queue being empty or not whether the car inserts itself in the queue. When the queue is empty, the car passes the light without delay and otherwise it inserts itself in the queue and waits. The car process ends when it is activated by the traffic light, which means that the car passes the light.

Listing 6.7: Car process

```
// Check the status of the trafic light.
if (model.status == model.RED) { // RED
    // Car enters the queue.
    model.carQueue.insert(this);

    // Wait in the queue.
    passivate();
} else { // GREEN
    // Enter the queue when the queue is not empty.
    if (!model.carQueue.isEmpty()) {
        // Enter the queue.
        model.carQueue.insert(this);

        // Wait in the queue.
        passivate();
    }
}
```

The implementation of the traffic light process is shown in Listing 6.8. Initially the status of the light is red, which is specified at the initialization of the simulation. The status is not declared within the traffic light process, because the car processes should also always have access to the current value. In each iteration, first it is checked whether a car can leave. Then, `slotsToChange` is decreased with one, which indicates the number of slots until the light has to change its colour. When the colour has to change, the tally that keeps track of the queue length is updated. At the end of each iteration the process is set to hold for one time unit, which is the remaining of a time slot.

Listing 6.8: Traffic light process

```
// Traffic light starts with red period.
int slotsToChange = model.nrOfRedSlots;

while (true) {
    // When the light is green, the first leaves.
    if (model.status == model.GREEN && !model.carQueue.isEmpty()) {
        Car firstCar = model.carQueue.first();
        model.carQueue.remove(firstCar);

        // Reactive the car for finishing its life cycle.
        firstCar.activate();
    }

    slotsToChange--;

    // Check whether the status has to change.
    if (slotsToChange == 0) {
        if (model.status == model.GREEN) { // Change to red.
            model.nrOfCarsEndGreen.update(model.carQueue.size());
            model.status = model.RED;
            slotsToChange = model.nrOfRedSlots;
        } else {
            model.nrOfCarsEndRed.update(model.carQueue.size());
            model.status = model.GREEN; // Change to green.
            slotsToChange = model.nrOfGreenSlots;
        }
    }

    // Wait until the next time slot starts.
    hold(new TimeSpan(1.0));
}
```

6.1.4 Output

DESMO-J automatically collect data about queues and shows the results in an HTML report. In this report the statistical data about queues, the tallies and other information about the simulation are included. Figure 6.1 shows a part of the report from the simulation of an M/M/c tandem queue. The network consists of three stations with at each station three servers. All servers have a service rate of 1.0 and customers arrive with an arrival rate of 2.5. The simulation ran for 100.000 time units.

We are interested in the mean waiting time at each station and in the sojourn time. Statistical data about the three customer queues can be found in the section Queues. We see that the average waiting times at the queues are 1.4485, 1.3921 and 1.4747. The theoretical value is 1.4045. The result of the simulation becomes more accurate when the simulation runs for a longer time or has multiple replications.

Tallies

Title	(Re)set	Obs	Mean	Std.Dv	Min	Max	Unit
System time	0.0000	250146	7.31993	3.8855	0.12038	33.41991	none

Queues

Title	Qorder	(Re)set	Obs	QLimit	Qmax	Qnow	Qavg.	Zeros	max.Wait	avg.Wait	refus.
Customer queue	FIFO	0.0000	250183	unlimit.	63	0	3.62391	73863	19.9927	1.4485	0
Idle server queue	FIFO	0.0000	73863	unlimit.	3	1	0.49471	1	5.6286	0.6697	0
Customer queue	FIFO	0.0000	250178	unlimit.	43	3	3.48275	74258	16.4084	1.3921	0
Idle server queue	FIFO	0.0000	74258	unlimit.	3	0	0.49517	1	5.3640	0.6668	0
Customer queue	FIFO	0.0000	250149	unlimit.	60	26	3.69017	73871	19.8282	1.4747	0
Idle server queue	FIFO	0.0000	73871	unlimit.	3	0	0.49363	0	5.4597	0.6682	0

Figure 6.1: Report M/M/c tandem queue

6.2 SSJ

In this section we explain the implementation details with the use of SSJ. The processes behave the same as in the implementation with the use of DESMO-J, but the synchronization and interaction between processes is handled differently. Again, the initialization of the simulation and the generation of processes that enter the system is the same in all models. Therefore, we discuss these parts of the program only for the first model.

6.2.1 M/M/c tandem queue

The initialization of the simulation of the first model is shown in Listing 6.9. In the method `simulate()` the simulation is started and the objects that are needed for the simulation are initialized. The class `SimProcess` provides the process scheduling tools. With the method `init()` the simulator to use processes is initialized. Next to the processes in the simulation, we also schedule one event that indicates the end of the simulation. This event is scheduled over a time given by the value of `simulationTime`. The class `Sim` provides the tools for the event-scheduling approach and is activated by calling the method `start()`.

The implementation of the customer generator process is shown in Listing 6.10. The implementation of this process is very similar to the implementation in DESMO-J. The arrival time of the customer is determined and a new customer process is activated at that time. Then, the generator waits until the customer process is activated before a new customer is created.

Listing 6.9: Initialization of the simulation

```
public void simulate(double simulationTime) {
    SimProcess.init();
    ...

    // Specify the simulation time.
    new EndOfSim().schedule(simulationTime);
    // Start the simulation.
    Sim.start();
}

class EndOfSim extends Event {
    public void actions() {
        Sim.stop();
    }
}
```

Listing 6.10: Customer generator process

```
while (true) {
    // Schedule arrival of the next customer.
    double nextArrivalTime = genArr.nextDouble();
    new Customer().schedule(nextArrivalTime);

    // Wait until the customer arrives before scheduling a new arrival.
    delay(nextArrivalTime);
}
```

The implementation of the customer process is shown in Listing 6.11. The arrival time of the customer is saved before the customer enters the first station. Then, at each station the customer requests one unit of the server resource. When this request can be handled, the customer is delayed for the time of the service. After the delay one unit of the resource is released again and the customer goes to the next station. When the customer leaves the last station, the tally that keeps track of the sojourn time is updated. Because we implemented the servers as resources, we do not need to implement a server process.

Listing 6.11: Customer process

```
// Specify the arrival time.
double arrivalTime = Sim.time();

// Customer goes through all stations.
for (int i = 0; i < nrOfStations; i++) {
    server[i].request(1);
    delay(genServ.nextDouble());
    server[i].release(1);
}

// Register the sojourn time of the customer.
sojournTime.add(Sim.time() - arrivalTime);
```

6.2.2 Polling system with gated service

The initialization and the customer generator process are the same as in the previous model. In the previous model it was not needed to implement a server process, but in the polling

model the server has some behaviour that needs to be described. The implementation of the customer process differs a lot from the implementation in DESMO-J. In the implementation with DESMO-J the customer simply added itself to the queue and waited for service. In the implementation with SSJ we do not have process queues available and we need to work with resources and conditions.

The implementation of the customer process is shown in Listing 6.12. We first get the arrival time of the customer. Then, the customer waits on the condition `arrival[station]` to become true by calling the method `waitFor()`. The condition becomes true for a short time when the server arrives at the corresponding station. Then, all customers that are waiting on that condition proceed. The processes encounter the server resource and request for a unit of the resource. The server is initialized with a capacity of one, since there is only one server that is able to serve one customer at the time. One customer request at the time is handled by the server. When the last customer is being served, the queue for the server resource is empty and the server is notified by setting the condition `finished` shortly to true.

The implementation of the server process is shown in Listing 6.13. Before the server starts serving customers it is specified that the server starts at station 0. When the server arrives at a station, the number of customers in the corresponding queue are counted. When there is at least one customer in the queue, the customers in that queue will be notified by setting the condition `arrival[position]` shortly to true. The server process waits until it is being notified that all customers are served. When all customers are served or when no customers had to be served, the server moves to the next station. The server is delayed until the switch-over time has passed. The position of the server is updated and in case the server is back at the first station the tally that keeps track of the mean cycle time is updated.

Listing 6.12: Customer process

```
// Specify the arrival time.
double arrivalTime = Sim.time();

// Wait for the server to arrive at the station.
arrival[station].waitFor();

// Server has arrived and the customers in the queue will be served.
server.request(1);

// The server starts serving.
waitingTime[station].add(Sim.time() - arrivalTime);
delay(genServ.nextDouble());

// Check whether all customers are served.
if (server.waitList().isEmpty()) {
    // Notify the server that he is done.
    finished.set(true);
    finished.set(false);
}

// The customer is served.
server.release(1);
```

Listing 6.13: Server process

```
// Specify the start conditions of the server.
int position = 0;
double startTime = 0.0;

while (true) {
    // Count the number of customers to be served.
    serveCustomers = arrival[position].waitList().size();

    // If there are customers waiting, serve them.
    if (serveCustomers > 0) {
        // Notify the customers that they will be served this round.
        arrival[position].set(true);
        arrival[position].set(false);

        // Wait until all customers are served.
        finished.waitFor();
    }

    // Switch the server.
    delay(switchTime);
    position = (position + 1) % nrOfStations;

        // If the server is back to the begin, register cycle time.
    if (position == 0) {
        cycleTime.add(Sim.time() - startTime);
        startTime = Sim.time();
    }
}
```

6.2.3 Discrete-time fixed-cycle traffic light

For the last model we have to describe the behaviour of the car and the traffic light. The traffic light process is responsible for keeping track of the time slots. This is done by setting a condition `nextSlot` shortly to true at the beginning of a new time slot. The traffic light is implemented as a resource with capacity one. This capacity indicates that at most one car can leave the queue during a time slot.

The implementation of the traffic light process is shown in Listing 6.14. Initially the status of the light is red, which is specified at the initialization of the simulation. At the start of each iteration, the light notifies a car process that the next time slot starts by shortly setting the condition `nextSlot` to true. Then, the light process is delayed for the time of one time slot. The value of `slotsToChange` is decreased by one, which indicates the number of slots until the light has to change its colour. When the colour has to change, the tally that keeps track of the queue length is updated and the `green` condition is either set to true or false. The traffic light process is scheduled in such a way that it first changes the colour of the light and then notifies a car that the next slot starts.

Listing 6.14: Traffic light process

```
// Traffic light starts with red period.
int slotsToChange = nrOfRedSlots;

while (true) {
    // Notify that the next time slot starts.
    nextSlot.set(true);
    nextSlot.set(false);

    // Duration of one time slot.
    delay(1.0);
    slotsToChange--;

    // Check whether the light has to change its status.
    if (slotsToChange == 0) {
        if (status == GREEN) { // Change to red.
            nrOfCarsEndGreen.add(light.waitList().size() +
                    green.waitList().size());
            status = RED;
            green.set(false);
            slotsToChange = nrOfRedSlots;
        } else { // Change to green.
            nrOfCarsEndRed.add(light.waitList().size() +
                    green.waitList().size());
            status = GREEN;
            green.set(true);
            slotsToChange = nrOfGreenSlots;
        }
    }
}
```

The implementation of the car process is shown in Listing 6.15. When a car arrives and the light is green and there is no car waiting in the queue or passing the light, the car passes the light without delay. Otherwise, the car requests the light for passing. Only the request of the first car in the queue is handled. Then, in case the light is red the car still has to wait before it can pass the light. When the light changes its colour to green it notifies the car by setting the condition `green` to true. The car that was waiting for the red light waits for the next slot to become true. Because the traffic light first notifies that the colour has changed to green and then that the next time slot starts, the first car leaves during the first green time slot.

Listing 6.15: Car process

```
// Check the status of the traffic light and the size of the queue.
if (status == GREEN && light.waitList().isEmpty() &&
        nextSlot.waitList().isEmpty()) {
    // Pass the light without delay
} else {
    // Get into the queue.
    light.request(1);

    // When the light is red, wait until it becomes green.
    if (status == RED) {
        green.waitFor();
    }

    // The car passes the light.
    nextSlot.waitFor();
    light.release(1);
}
```

6.2.4 Output

SSJ automatically collects data about the waiting lists for resources and conditions. Whether the statistical results of this data are printed should be specified for each waiting list. The same holds for additional tallies that are created. Figure 6.2 shows the output of the simulation of an M/M/c tandem queue with the same settings as used for the output generated with DESMO-J in the previous section.

There are three resources named server, each corresponding to one of the stations. We can see that the average waiting times are 1.473, 1.470 and 1.454. In contrast to the output of DESMO-J the service times are also included. Although we could compute the system time by simply adding the waiting times and service times at each station, we introduced a tally.

The output of a simulation is not always generated. SSJ has some difficulties with the synchronization and handling of processes. When there are too many processes active at the same time, so a lot of concurrent threads, the program executes code that should not be possible. We encountered the following three errors:

```
calling delay() for a process not in EXECUTING state
calling resume() for a dead process
trying to release more units of a Resource than the process currently holds
```

```
REPORT ON RESOURCE : Server
   From time :     0.00   to time :  100000.00
                    min         max      average  standard dev.  nb. obs.
   Capacity           3           3        3.000
   Utilization        0           3        2.505
   Queue Size         0          79        3.692
   Wait           0.000      29.260        1.473       2.130     250543
   Service      3.6E-7      14.319        1.000       0.999     250540
   Sojourn      3.6E-7      33.440        2.473       2.354     250540

REPORT ON RESOURCE : Server
   From time :     0.00   to time :  100000.00
                    min         max      average  standard dev.  nb. obs.
   Capacity           3           3        3.000
   Utilization        0           3        2.513
   Queue Size         0          48        3.684
   Wait           0.000      21.052        1.470       2.016     250539
   Service      8.6E-6      12.077        1.003       1.002     250536
   Sojourn      1.0E-5      23.190        2.473       2.252     250536

REPORT ON RESOURCE : Server
   From time :     0.00   to time :  100000.00
                    min         max      average  standard dev.  nb. obs.
   Capacity           3           3        3.000
   Utilization        0           3        2.507
   Queue Size         0          55        3.642
   Wait           0.000      16.434        1.454       1.984     250531
   Service      2.7E-6      12.317        1.001       0.999     250528
   Sojourn      2.7E-6      23.060        2.454       2.223     250528

REPORT on Tally stat. collector ==> sojourn time
    num. obs.       min          max        average     standard dev.
    250528        0.027       42.801         7.401        4.001
```

Figure 6.2: Output M/M/c tandem queue

Chapter 7

Results

In this chapter we discuss the implementations without framework and the two implementation with framework. We compare the implementations based on the following criteria: performance, understandability, implementation effort, scalability, reliability and data collection. For the criteria performance and scalability running times are measured. For each result we ran the simulation ten times and took the average of these observations. All individual observations and the exact mean value is listed in appendix A.

Performance
We measured the performance by simulating the M/M/c tandem queue for a million time units. We chose the settings in such a way that the number of customers in the system remains small. There are three stations with at each station three servers. The arrival rate of customers is 2.5 and the service rate is 1.0. With these settings the FES is filled with a few events or the number of threads during the simulation is low in case of the implementation without framework and with frameworks, respectively. The performance with a high number of customers is measured when we compare the scalability of the implementations. Figure 7.1 shows the running times of the three implementations. We combined the results in one graph such that the relative performance is clearly visible. When we look at this graph, we see that the performance significantly decreases when frameworks are used. When a million time units are simulated, the performance when SSJ is used is decreased with about a factor 60 and when DESMO-J is used even with about a factor 250.

Understandability
This criterion and also the criterion implementation effort are subjective. We believe that the understandability of a program that is implemented with the process-interaction approach is higher than with the event-scheduling approach. The lines of code that are needed is less and different parts of the program are naturally separated. It is easier to understand what the behaviour of a customer and a server is, than to understand what happens and who are involved in a particular event. So, we believe that the understandability of the implementation with the use of one of the frameworks is higher than the implementation without framework. Both frameworks have each there own methods for the process interaction and synchronization. DESMO-J uses queues in which processes are inserted and removed and SSJ uses resources and conditions. Both frameworks are best understood in one of the models. For the implementation of the M/M/c tandem queue SSJ only had to implement the customer

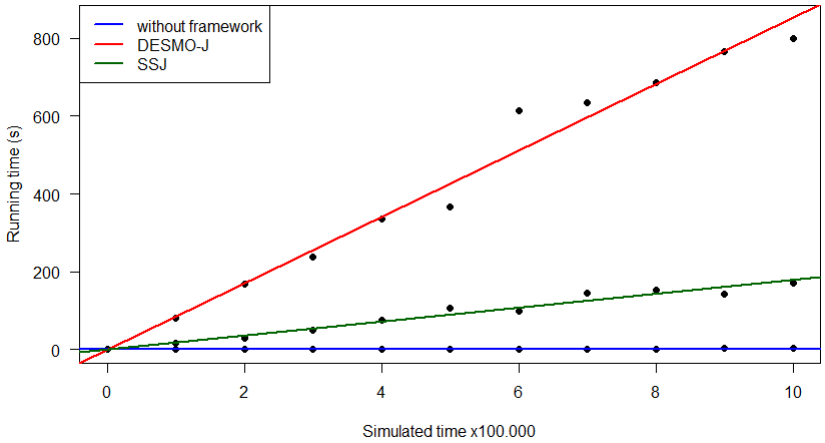

Figure 7.1: Performance of the implementations

process and needed only a few lines of code. With the use of DESMO-J the server process also needed to be implemented and more lines of code were used. On the other hand, the implementation of the polling system with DESMO-J is much clearer than with SSJ.

Implementation effort

This criterion mostly coincides with the understandability. When a program is more understandable, the implementation effort is probably also less. The implementation effort without using a framework is the highest. All standard classes such as queues, the FES and the classes for the data collection are not included in Java itself. Both frameworks provide enough classes to implement the models without complex constructions. Although the understandability of both frameworks is about the same, we believe that the implementation effort is higher for DESMO-J than for SSJ. To illustrate this, we consider the example of a server process in the M/M/c tandem queue. In DESMO-J each server process needs to be activated separately and it needs to be specified at what station it should be positioned. For this we create an idle server queue at each of the stations. The resulting code is clearly understandable, but it takes some effort to create the queues and use them. In SSJ the server processes are only described by a resource at each station. Also the initialization is faster to implement in SSJ than it is in DESMO-J.

Scalability

We tested the scalability of the implementations by simulating 100.000 time units of the tandem queue and the polling system for different number of stations. With the use of Little's law, we compute the mean number of customers in the system. For the tandem queue, this results in the formula:

$$\mathbb{E}[C] = \lambda \mathbb{E}[S] = \lambda N(\mathbb{E}[W] + \frac{1}{\mu}). \tag{7.1}$$

The mean waiting time at a station, denoted by $\mathbb{E}[W]$, does not depend on N. When we scale the number of stations in the tandem queue, the number of active processes scales linear with the number of stations. For the simulation of this model we choose the number of servers at each station equal to three with a service rate of 1.0 and the customer arrival rate equal

to 2.5. When we fill in these values, the formula for the mean number of customers in the system is:

$$\mathbb{E}[C] = 6.011N. \tag{7.2}$$

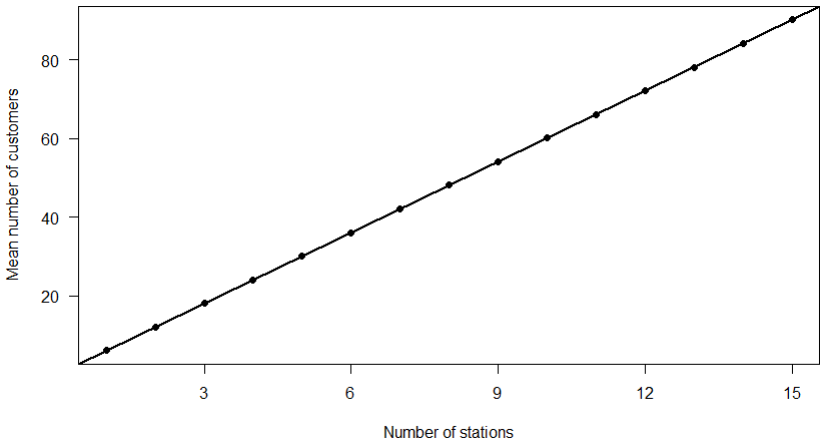

Figure 7.2: Mean number of customers in the tandem queue

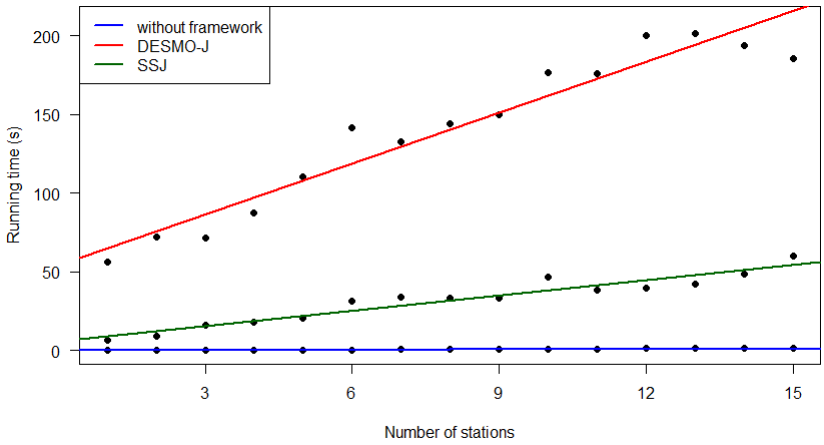

Figure 7.3: Running times of the scaled tandem queue model

Figure 7.2 shows the mean number of customers in the system for different number of stations. In Figure 7.3 the results are shown of the running times for the different implementations. We see that the running time increases linear with the number of stations. The difference between the implementations without and with frameworks is that the running times for more stations becomes more unpredictable. The number of active processes in this simulation is limited, so we also tested the scalability when we increase the number of concurrent processes significantly. We do this by simulating the polling system.

For the simulation of the polling system we choose the switch-over time equal to 1.0, the arrival rates equal to 1.0 and the service rate equal to the number of stations plus one. Then,

the stability condition that $\frac{N\lambda}{\mu} < 1$ is satisfied, but it goes to one when the number of station increases. Again with the use of Little's law, we obtain for the mean number of customers in the system:

$$\mathbb{E}[C] = \lambda N \mathbb{E}[W] = \lambda N \left(\frac{\rho}{1-\rho} \left(\frac{1}{\mu} + s \right) + \frac{Ns}{2} + \frac{Ns\rho}{2(1-\rho)} \left(1 - \frac{1}{N} \right) \right). \tag{7.3}$$

When we fill in the values $s = 1.0$, $\lambda = 1.0$ and $\mu = N + 1$, we get:

$$\mathbb{E}[C] = N \left(\frac{\frac{N}{N+1}}{1 - \frac{N}{N+1}} \left(\frac{1}{N+1} + 1 \right) + \frac{N}{2} + \frac{N\frac{N}{N+1}}{2(1 - \frac{N}{N+1})} \left(1 - \frac{1}{N} \right) \right). \tag{7.4}$$

Simplifying this formula results in:

$$\mathbb{E}[C] = \frac{N^3}{2} + N^2 + \frac{N^2}{N+1}. \tag{7.5}$$

When we increase the number of stations, the number of customers that is on average in the system increases cubically. Figure 7.4 shows the mean number of customers in the system for different number of stations. Figure 7.5 shows the results of the running times for the different implementations. In case no framework is used the running time increases linear with the number of stations. The cubic increase of the number of customers in the system does not result in a cubic increase of the running time. The running time in comparison with the tandem queue model is even faster. The running times of both the implementations with DESMO-J and SSJ increase with the number of active processes in the system. For these implementations, a cubic function fits the increase in running time best.

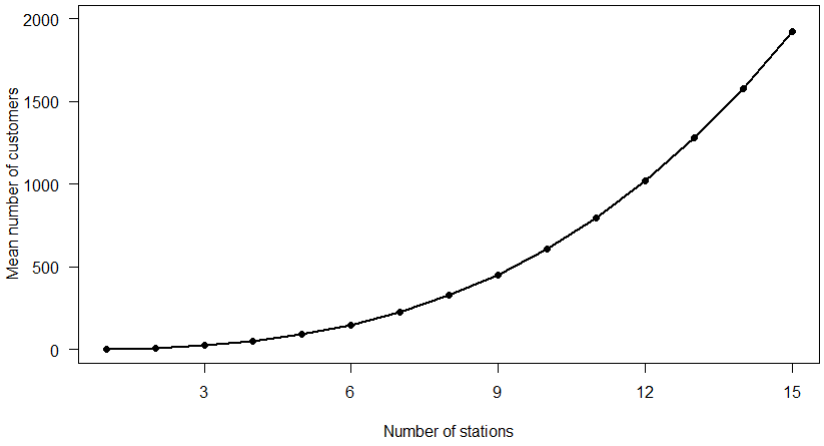

Figure 7.4: Mean number of customers in the polling system

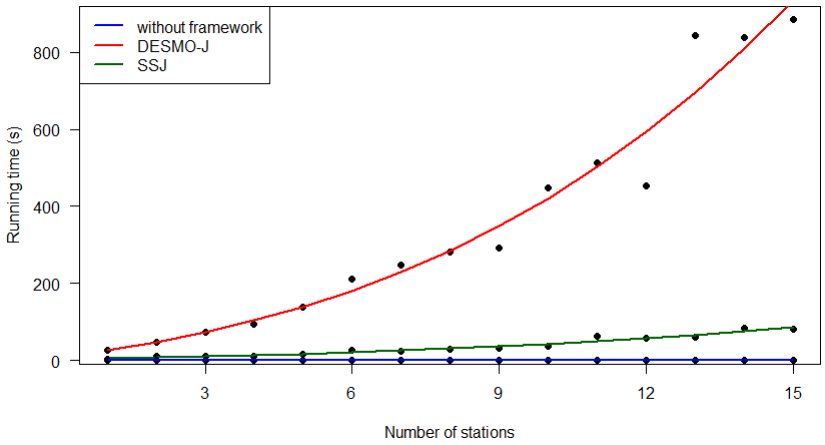

Figure 7.5: Running times of the scaled polling system model

Important to note is that increasing the number of stations in the polling system to more than 15 with these settings is not possible when the frameworks are used. Then, the number of concurrent processes exceeds the maximum number of threads. In the case of SSJ, the running times in comparison with the previous model are approximately equal. When DESMO-J is used, the running time increases with about a factor 5 for 15 number of stations. Since both frameworks are not able to handle the situation in which a lot of concurrent processes are active, the implementations with framework score low on the scalability criteria.

Reliability
The reliability of the implementation without framework and with DESMO-J is high. There occurred never problems during the simulation and the output is always as expected. As already mentioned in Chapter 6 the use of SSJ resulted in problems with the thread synchronization. These problems did not occur when the number of concurrent processes is limited, but occurred frequently when this number increases. For example, the simulations for which we measured the scalability of the polling system failed about half of the times when the number of stations was more than 10.

Data collection
Automatic data collection is not provided in Java, so we can not give a score to this for the implementation where no framework is used. Methods for data collection and for the output of the statistical results are provided in both frameworks. The report that is generated in DESMO-J is more extensive than the output that SSJ generates. Also other possibilities for data collection are provided in DESMO-J. These include counters, accumulates, histograms, regressions and time series. SSJ only provides the possibility to collect the data about the queues of resources and conditions.

The results are summarized in Table 7.1. The number of pluses is an indication of how well the criterion is satisfied. The performance is best for the implementation without framework, then with SSJ, and worst with DESMO-J. The understandability is worst when no framework is used. We gave three pluses to both frameworks for the understandability, because it is subjective which one is preferred. The implementation effort is believed to be

higher for DESMO-J than it is for SSJ. Scalability scores low for both frameworks, because the simulation may only contain a maximum number of concurrent processes. Without the use of a framework there is no such limit. The reliability is high for both the implementation without framework and with DESMO-J. Only with the use of SSJ errors occurred frequently during the simulation. For the data collection DESMO-J scores the highest, because there are more methods provided for this purpose than in SSJ.

criterion	**without framework**	**DESMO-J**	**SSJ**
performance	+++	+	++
understandability	+	+++	+++
implementation effort	+	++	+++
scalability	+++	+	+
reliability	+++	+++	+
data collection	0	+++	++

Table 7.1: Comparison of the three simulation programs

Counting the number of pluses results in DESMO-J to be the best way of implementing a simulation. In comparison with SSJ only the performance is less. However, it is much more reliable than SSJ. A framework with high performance but with low reliability should not be preferred over a framework that is reliable with a lower performance. Only at understandability and implementation effort the implementation without framework scores low. However, it should be kept in mind that learning a framework also takes some time. Especially performance is an important criterion for simulation and the computation of results can be implemented by hand. So, when multiple replications or a long simulation time is needed an implementation without framework is the best choice.

Chapter 8

Conclusion

In this report we studied the best way to implement discrete event simulations in Java. We did this by means of implementing three different models. We implemented the models with the use of the event-scheduling approach and the process-interaction approach. The event-scheduling approach is implemented with only the standard classes provided in Java and the process-interaction approach with frameworks. First, we explained the implementation details of the event-scheduling approach. We used the standard random number generator provided in Java and the `ArrayList` data structure for the Future Event Set. A performance improvement can be achieved by using an alternative random number generator and data structure. The Mersenne Twister for the random number generator already improves the performance. An even better improvement can be obtained by implementing the `PriorityQueue` as alternative for the `ArrayList` of the FES. The improvement for the implementation of the M/M/c tandem queue with these alternative implementations is about 21%.

Next, we explained the implementation details of the process-interaction approach. For the implementation of the models we used the two frameworks that seemed to offer the best functionality: DESMO-J and SSJ. The frameworks provide different methods for process interaction and synchronization. In order to investigate whether the frameworks are a good support for the implementation of a discrete event simulation, we ranked the frameworks on several criteria. Whether the implementation effort decreases and the understandability of the program increases is subjective. Based on these criteria no conclusion can be drawn which of the two frameworks is best. We do conclude that the implementation with the use of one of the frameworks is more understandable and requires less implementation effort than the implementation without framework. The frameworks are also compared on the objective criteria performance, scalability, reliability and data collection. We summarized the scores for the criteria in Table 7.1 presented in Chapter 7. After we added the scores for all criteria, the framework DESMO-J turned out to be the best choice.

We conclude that DESMO-J is the best framework to use. However, a process-interaction approach limits the scalability of a simulation, because the number of concurrent threads is limited. Also the performance is decreased significantly when the process-interaction approach is used. When a long simulated time or multiple replications are needed, the performance of the program becomes more important. Then, a process-interaction approach should not be preferred over the event-scheduling approach.

Bibliography

[1] Arena simulation software. Internet: www.arenasimulation.com, [Mar. 20, 2017].

[2] I.J.B.F. Adan, J.A.C. Resing. Lecture notes, Topic: Queueing Systems. Department of Mathematics and Computing Science at Eindhoven University of Technology, 2015.

[3] O. J. Boxma, W. P. Groenendijk. Pseudo-conservation laws in cyclic-service systems. *Journal of Applied Probability*, vol. 24(4), pp. 949964, 1987.

[4] J.S.H. van Leeuwaarden. Delay Analysis for the Fixed-Cycle Traffic-Light Queue. *Transportation Science*, vol. 40(2), pp. 189-199, 2006.

[5] M. Matsumoto, T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, vol. 8(1), pp. 3-30, 1998.

[6] S. Andradóttir, K.J. Healy. Silk: a Java-based process simulation language. *Proceedings of the 1997 Winter Simulation Conference*, 1997, pp. 475-482.

[7] A.H. Buss, K.A. Stork. Discrete event simulation on the world wide web using Java. *Proceedings of the 1996 Winter Simulation Conference*, 1996, pp. 780-785.

[8] P.H.M. Jacobs, N.A. Lang, A.Verbraeck. D-SOL; a distributed Java based discrete event simulation architecture. *Proceedings of the 2002 Winter Simulation Conference*. 2002, pp. 793-800.

[9] Download link DESMO-J. Internet: www.desmoj.sourceforge.net, [Feb. 19, 2017].

[10] J. Göbel, P. Joschko, A. Koors, B.Page. The discrete event simulation framework DESMO-J: review, comparison to other frameworks and latest development. *Proceedings 27th European Conference on Modelling and Simulation*, 2013, pp. 100-109.

[11] Download link SSJ. Internet: simul.iro.umontreal.ca/ssj-2, [Mar. 06, 2017].

[12] P. L'Ecuyer, L. Meliani, J. Vaucher. SSJ: a framework for stochastic simulation in Java. *Proceedings of the 2002 Winter Simulation Conference*. 2002, pp. 234-242.

[13] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1), pp. 159-164, 1999.

Appendix

A Performance Results

In this appendix we present the individual observations of the mean running time for different implementations. The first table presents the running times of the improvements for the implementation without framework as discussed in Chapter 4. The other tables present the running times of the different implementations to test the performance and scalability as discussed in the Chapter 7.

implementation	mean running time (ms)	observations
1	35994	35409, 36429, 36453, 36854, 35522, 36374, 35030, 35839, 36282, 35749
2	34337	33838, 34158, 34648, 33877, 34516, 34532, 34804, 34325, 34196, 34475
3	29831	30519, 29399, 29642, 30507, 30061, 29412, 30044, 29541, 29515, 29673
4	28496	28789, 28884, 28590, 28163, 28333, 28547, 28497, 28646, 28340, 28173

Table 1: Running times different implementations

simulated time	mean running time (ms)	observations
100.000	308	297, 313, 297, 297, 297, 344, 297, 313, 313, 313
200.000	591	579, 594, 594, 593, 593, 578, 594, 593, 594, 594
300.000	875	860, 906, 859, 859, 875, 875, 875, 859, 859, 922
400.000	1147	1140, 1157, 1141, 1125, 1140, 1141, 1156, 1141, 1141, 1187
500.000	1439	1422, 1437, 1432, 1468, 1422, 1406, 1503, 1422, 1422, 1453
600.000	1715	1703, 1703, 1703, 1719, 1704, 1703, 1703, 1704, 1781, 1725
700.000	2007	1984, 2000, 2047, 1984, 1968, 2178, 1984, 1953, 1984, 1984
800.000	2261	2251, 2266, 2265, 2281, 2265, 2282, 2250, 2266, 2235, 2250
900.000	2566	2531, 2563, 2532, 2531, 2547, 2578, 2570, 2587, 2641, 2578
1.000.000	2874	2812, 2848, 2860, 2984, 2852, 2860, 2906, 2828, 2875, 2913

Table 2: Running times tandem queue without framework for different simulated times.

simulated time	mean running time (ms)	observations
100.000	80108	120001, 69899, 87195, 73553, 74620, 69379, 73765, 71644, 77127, 83896
200.000	167219	177900, 150687, 186657, 159548, 153802, 166205, 188172, 163199, 159522, 166496
300.000	237346	219839, 270373, 233837, 230543, 225213, 269424, 265874, 233739, 207336, 217282
400.000	337156	317823, 295586, 308493, 324361, 318290, 308519, 354690, 387146, 440207, 316440
500.000	367681	381202, 371880, 385061, 358530, 411848, 385345, 348886, 344452, 346825, 342779
600.000	613166	660987, 781795, 458706, 561669, 470313, 496181, 511385, 995342, 654915, 540370
700.000	634769	730237, 497208, 572121, 578949, 847410, 776740, 590181, 569220, 577679, 607942
800.000	686585	559626, 565263, 663550, 691403, 598604, 823372, 666978, 625971, 855038, 816046
900.000	765610	753212, 804656, 840722, 701905, 675583, 788283, 765832, 844048, 772774, 709087
1.000.000	800645	713953, 945599, 821005, 708928, 792420, 885554, 722511, 814841, 759616, 842026

Table 3: Running times tandem queue with DESMO-J for different simulated times.

simulated time	mean running time (ms)	observations
100.000	16778	14922, 16923, 19047, 21234, 12282, 11187, 10578, 21376, 17093, 23141
200.000	29461	31913, 27335, 49735, 23661, 30750, 21438, 31375, 34001, 21437, 22969
300.000	50569	50504, 55750, 44891, 44247, 44063, 50626, 44803, 40394, 64789, 65620
400.000	75359	90598, 56316, 78391, 68799, 81503, 88292, 85191, 66738, 68983, 68781
500.000	106586	64180, 104250, 60900, 132124, 122686, 89193, 133052, 133312, 76778, 149380
600.000	99489	65287, 65316, 98270, 68833, 69019, 115248, 100384, 128649, 106274, 177610
700.000	144479	134283, 164744, 182430, 144924, 137749, 141233, 143860, 106408, 131693, 157467
800.000	153430	172509, 167825, 154486, 170279, 166119, 126230, 164823, 134011, 179662, 98355
900.000	142397	131858, 149392, 195902, 142131, 118817, 119254, 140834, 103369, 175704, 146706
1.000.000	171958	214977, 141738, 131308, 147704, 191580, 200237, 200071, 173470, 148497, 170002

Table 4: Running times tandem queue with SSJ for different simulated times.

number of stations	mean running time (ms)	observations
1	128	140, 125, 125, 125, 140, 125, 172, 109, 109, 109
2	217	219, 203, 218, 219, 219, 218, 219, 219, 219, 219
3	313	344, 313, 297, 312, 312, 328, 297, 312, 313, 297
4	402	406, 391, 406, 391, 407, 407, 406, 391, 407, 406
5	497	500, 484, 500, 485, 500, 484, 531, 485, 500, 500
6	558	672, 562, 547, 547, 547, 531, 547, 547, 531, 547
7	672	766, 719, 656, 687, 657, 656, 640, 657, 640, 641
8	775	812, 781, 750, 766, 766, 765, 828, 750, 766, 766
9	939	1109, 906, 1094, 891, 906, 891, 906, 890, 907, 890
10	1044	1031, 1016, 1031, 1016, 1110, 1046, 1031, 1047, 1047, 1062
11	1161	1266, 1125, 1125, 1281, 1109, 1141, 1187, 1141, 1109, 1125
12	1325	1422, 1328, 1187, 1188, 1266, 1328, 1375, 1187, 1282, 1687
13	1380	1469, 1328, 1531, 1422, 1312, 1297, 1297, 1297, 1297, 1547
14	1427	1515, 1422, 1485, 1390, 1407, 1406, 1422, 1406, 1406, 1407
15	1613	1594, 1641, 1578, 1608, 1678, 1610, 1547, 1625, 1640, 1609

Table 5: Running times tandem queue without framework for different number of stations.

number of stations	mean running time (ms)	observations
1	56541	63657, 54716, 54395, 55286, 56736, 62014, 55491, 54002, 54323, 54791
2	72437	61329, 79098, 62593, 91935, 62811, 71825, 63144, 68857, 92047, 70733
3	71756	69643, 70503, 71773, 70258, 71445, 74186, 70365, 73444, 76557, 69583
4	87432	80494, 85572, 88481, 77225, 76534, 89432, 134719, 83480, 79238, 79146
5	110554	130188, 123949, 89354, 94981, 104855, 103909, 117328, 121876, 129860, 89236
6	141411	162126, 107446, 142192, 141558, 162521, 89339, 228849, 145122, 140568, 94393
7	132711	104457, 129824, 108780, 178931, 108536, 104903, 242731, 108237, 112005, 128705
8	143908	115481, 116312, 140597, 122926, 145983, 264264, 137799, 144490, 113126, 138099
9	150122	120272, 144405, 152225, 167424, 120028, 185808, 154756, 167546, 163739, 125021
10	176829	169101, 228080, 182747, 146014, 156422, 148532, 156112, 228719, 175731, 176828
11	175908	154270, 157447, 142932, 155375, 189846, 185159, 198094, 249103, 176481, 150371
12	200155	172583, 225217, 180503, 189259, 179295, 156294, 251603, 163165, 176187, 307444
13	201604	167041, 177613, 188244, 238138, 275250, 193734, 238017, 178750, 190427, 168829
14	194015	294581, 180047, 179340, 180301, 163665, 193153, 190334, 203331, 186855, 168540
15	185392	191890, 202221, 205188, 185104, 172192, 171828, 210988, 171377, 176189, 166938

Table 6: Running times tandem queue with DESMO-J for different number of stations.

number of stations	mean running time (ms)	observations
1	6341	5335, 11282, 6287, 5187, 5268, 6164, 5773, 6300, 5318, 6493
2	9143	8407, 11078, 9309, 9307, 9031, 8313, 8298, 8895, 9563, 9232
3	15888	16031, 18318, 10904, 15914, 14338, 14645, 25266, 11617, 10746, 21104
4	18149	28048, 18053, 17314, 23518, 17789, 13469, 13059, 18120, 19216, 12900
5	20579	27298, 16933, 17889, 17339, 15555, 22662, 24057, 17147, 21276, 25637
6	31365	31240, 18649, 54107, 43880, 28469, 30500, 39515, 19209, 26670, 21406
7	34075	40766, 34637, 43720, 35453, 27716, 30801, 23594, 27382, 28391, 48294
8	33383	32530, 49594, 30719, 45438, 43469, 31922, 28126, 24141, 24031, 23860
9	33570	33869, 50126, 34641, 34656, 31969, 26829, 35031, 31329, 29906, 27344
10	46674	38756, 44243, 42853, 46298, 34653, 34459, 70317, 60100, 63457, 31600
11	38643	52976, 31860, 45797, 44751, 31625, 39281, 34798, 34203, 35438, 35703
12	39611	40578, 38376, 41750, 37172, 40376, 39656, 34360, 41594, 39251, 43000
13	42307	48023, 57079, 40906, 40141, 39532, 40016, 39313, 37328, 43454, 37282
14	48450	73775, 59033, 40989, 41736, 48569, 45160, 42579, 42319, 43394, 46944
15	59817	57437, 64806, 61304, 53158, 58782, 57313, 69233, 52124, 55992, 68019

Table 7: Running times tandem queue with SSJ for different number of stations.

number of stations	mean running time (ms)	observations
1	62	62, 62, 62, 62, 63, 62, 62, 62, 63, 63
2	100	94, 110, 93, 93, 109, 94, 93, 110, 109, 94
3	155	172, 188, 125, 188, 140, 141, 141, 172, 141, 139
4	191	188, 188, 187, 219, 187, 188, 188, 188, 187, 187
5	230	219, 235, 235, 235, 219, 234, 219, 234, 234, 235
6	245	266, 250, 234, 235, 234, 234, 235, 250, 234, 281
7	298	297, 296, 422, 282, 281, 281, 281, 282, 281, 281
8	341	343, 454, 328, 328, 328, 312, 329, 328, 328, 328
9	376	391, 397, 374, 375, 375, 359, 375, 359, 375, 375
10	483	515, 453, 500, 453, 625, 453, 453, 453, 453, 468
11	494	594, 469, 484, 484, 469, 468, 516, 500, 487, 468
12	519	547, 531, 503, 516, 515, 516, 515, 516, 516, 515
13	579	594, 578, 593, 567, 563, 609, 562, 576, 566, 578
14	657	640, 649, 652, 610, 657, 640, 672, 813, 609, 625
15	733	797, 703, 703, 719, 719, 703, 703, 812, 718, 750

Table 8: Running times polling system without framework for different number of stations.

number of stations	mean running time (ms)	observations
1	25624	21998, 23375, 22360, 22062, 22250, 26062, 48865, 23160, 23327, 22777
2	47293	76884, 43039, 43000, 43407, 42880, 43114, 51813, 42827, 42722, 43244
3	73989	121952, 64783, 64918, 67964, 66814, 63975, 85661, 65263, 68498, 70061
4	93351	95386, 104219, 93212, 92339, 92446, 97658, 93402, 89909, 87696, 87241
5	137871	173811, 114410, 124251, 146985, 133501, 131426, 146135, 135220, 140738, 132230
6	212452	292796, 313666, 151806, 341734, 222369, 162654, 170565, 159195, 152640, 157099
7	248285	290543, 238970, 227700, 295211, 217645, 230022, 274175, 213709, 266893, 227977
8	281247	325812, 254488, 351445, 202761, 213454, 252288, 374473, 275345, 311169, 251231
9	291163	423845, 307603, 318283, 289766, 272486, 291753, 273488, 232309, 266222, 235873
10	447567	387475, 491223, 405562, 669475, 784672, 421341, 302445, 300786, 338179, 374511
11	512311	440745, 474037, 606262, 598731, 369736, 425353, 474425, 478385, 701858, 553573
12	452792	539245, 495404, 443965, 409110, 456008, 391340, 517501, 390601, 422738, 462008
13	844480	922870, 681042, 1160071, 760573, 1028566, 809971, 776284, 871628, 765990, 667806
14	837835	607512, 853976, 861436, 739905, 987488, 740113, 822185, 1088433, 974988, 702313
15	883984	868802, 915641, 1232495, 1058241, 762270, 821793, 714549, 906161, 825310, 734575

Table 9: Running times polling system with DESMO-J for different number of stations.

number of stations	mean running time (ms)	observations
1	4469	8250, 5734, 3359, 3516, 6297, 3063, 3266, 5203, 2969, 3031
2	9797	16531, 14376, 7125, 6250, 12860, 5782, 7219, 10719, 5437, 11672
3	11575	15419, 11567, 8315, 7938, 16434, 8388, 14141, 13486, 8963, 11098
4	12090	11670, 11037, 10761, 10928, 16438, 10954, 11455, 11318, 11387, 14948
5	17351	20544, 13741, 16377, 13606, 14016, 13344, 14727, 17710, 25016, 24425
6	26551	23734, 42688, 32798, 16230, 29625, 27203, 19251, 31141, 25094, 17750
7	24669	20473, 20489, 18964, 37502, 28451, 24172, 20235, 22140, 29266, 25001
8	28508	27170, 22381, 22246, 24458, 25082, 22516, 39846, 35889, 28253, 37236
9	30795	29319, 27608, 29210, 26734, 30277, 37659, 28600, 34338, 38041, 26160
10	35759	34523, 38416, 41351, 32966, 32783, 34741, 36436, 32925, 37776, 35669
11	62326	70142, 63140, 46077, 49885, 78339, 68140, 59775, 51978, 53219, 82568
12	58756	50787, 64731, 53742, 59648, 58488, 85552, 60288, 40690, 50068, 63567
13	60180	61755, 60914, 58479, 56141, 64654, 54352, 59707, 79160, 51282, 55353
14	83174	96794, 79924, 72436, 83027, 58324, 67283, 86662, 106589, 106638, 74062
15	81812	97553, 106125, 76554, 74069, 70906, 63770, 71647, 102951, 75106, 79435

Table 10: Running times polling system with SSJ for different number of stations.