1. Discrete Event Simulation with SimPN

The goal of business process simulation is to write a computer program that behaves similarly to your business process with respect to the relevant properties that you want to measure. Usually, these include:

- timing properties, such as the waiting times of clients and the total time it takes for the process to complete for a client;
- costs, such as the total cost of running the process or the cost of running the process for a particular client; and
- the number of resources that work on the process.

Once a simulation has been built that indeed behaves similarly to the actual business process in real life, changes can be made to the business process to see what the impact of the change would be on the relevant properties. This is also known as what-if analysis. For example, we could create a simulation of a help-desk process and subsequently do a what-if analysis to see if we can reduce the number of help-desk employees, while still being able to respond to customers within 24 hours. We could also do a what-if analysis on what would happen to the occupancy rate of our employees and the time it takes to answer our customers if we do more work in parallel.

This chapter presents a specific form of simulation called discrete event simulation, which can be used to model the state of a system and the events that change the state of the system. The system can be a factory in which events happen such as starting to manufacture a product on a machine, completing the production on a machine, or a machine breaking down. The system can also be a business process, in which customers arrive, are served by employees, and leave again.

1.1 The Basics

In a discrete event simulation, the state of a system changes when an event happens. Nothing but events can change the system. Consequently, a discrete event simulation can be completely described by modeling what makes up the state of the system, which events can happen in the system, in which state they can happen, and how they change the state of the system into a new state (in which new events can happen). We consider events ‘atomic’, meaning that events do not take time and that, when an event happens, it happens as a whole - uninterrupted by other events. After all it does not take time, so no other events can happen in between, because there is no ‘in between’. It is possible that multiple events happen at the same time. In that case one event needs to be executed before the other. Which of the events happens before the other is typically undefined and therefore random. However, this is important, because one event can change the state of the system in such a way that the other cannot happen anymore. For example, if two customers can start to be served at the same moment, but there is only one resource available to help them, when the event of starting to serve one customer happens, the other customer will have to wait.

As an example of a discrete event simulation, consider a system consisting of a queue and a cashier at a cash register. The state of the system is described by the number of customers in the system and whether the cashier is busy serving a customer. The state can change due to the events of a new customer arriving in the queue, the cashier starting to service the customer, or the cashier completing their service.

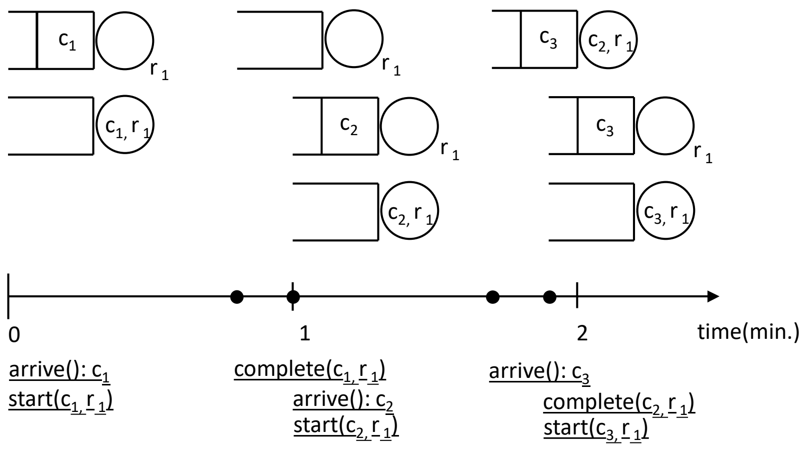

Figure 1.1: an example timeline of a discrete event simulation.

For the example, let us assume that one customer arrives on average each minute (according to a Poisson process) and it takes on average 0.8 minutes to serve a customer (exponentially distributed). Figure 1.1 shows a possible timeline of events that matches this description. It also shows how the state of the system, represented as a queue, develops over the timeline. The cashier resource is represented as a circle and the queue of waiting customers is represented as the rectangle in front of the resource. The first customer, c_1, arrives at time $t = 0$. Now, the state of the systems changes to one customer waiting in the system. No customer is being served, so the cashier resource is drawn outside of its circle. Since the cashier, labeled resource r1, is available, we can immediately start serving the customer, which leads to a state of the system where customer c_1 is being served. In the timeline, serving this customer takes exactly 0.8 minutes, after which the cashier completes their service, leading to a system where there is no waiting customer and no customer being served. The next customer arrives at $t = 1$ and can again start to be served immediately. The next customer, c_3, arrives at $t = 1.7$. However, at that time customer c_2 is still being served, so c_3 will have to wait in line. And so on.

The timeline from Figure 1.1 is an example of a timeline for a system in which events have already happened. However, initially no events will have happened in the system. Events will happen over time, when the state of the system allows for them. When they do, they will change the state of the system, such that other events can happen. Note that earlier events must always happen first, so events happen in the order in which they appear on the timeline. Time also progresses as events happen. So, initially the time will be 0, and at each point in the simulation the earliest event that can happen will happen and the time of the simulation will be the time moment of that event. This principle prevents inadvertent time travel: when an event that happens at a later time happens before an event that at an earlier time. For example, when a customer starts to be served before it arrives.

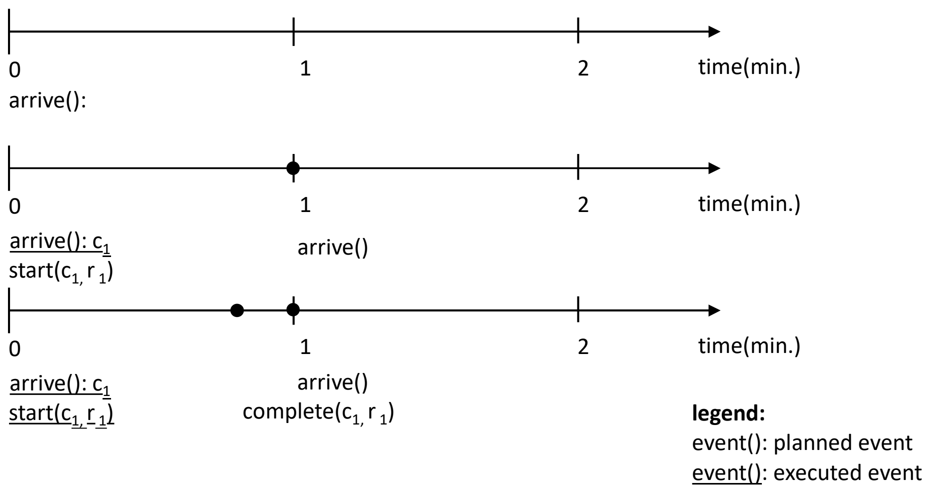

Figure 1.2: the example timeline built gradually.

Figure 1.2 shows how the timeline is built up gradually after each event. It distinguishes between events that are planned to happen and events that have happened happen. The latter events are underlined. The former are not. Initially, only the arrival of a customer can happen. When it happens, it enables the start of serving the customer that just arrived. It also enables the arrival of the next customer. Now these two events can happen. In the figure, starting to serve the customer happens next, because it is the earlier event. This enables the completion of serving that customer. So now the completion of serving the customer and the arrival of the next customer are enabled. Completing to serve the customer will happen next, because it is the earlier event. When it happens, the current simulation time will also be updated to $t = 0.8$, because this is the time at which the event happened.

We need one last ingredient to fully understand the basics of discrete event simulation. Remember that whether an event can happen depends on the state of the system and that when an event happens it changes the state of the system. To be able to represent that, we introduce variables that represent the state of the system.

Figure 1.3 shows a timeline of the situation after the first customer has arrived and

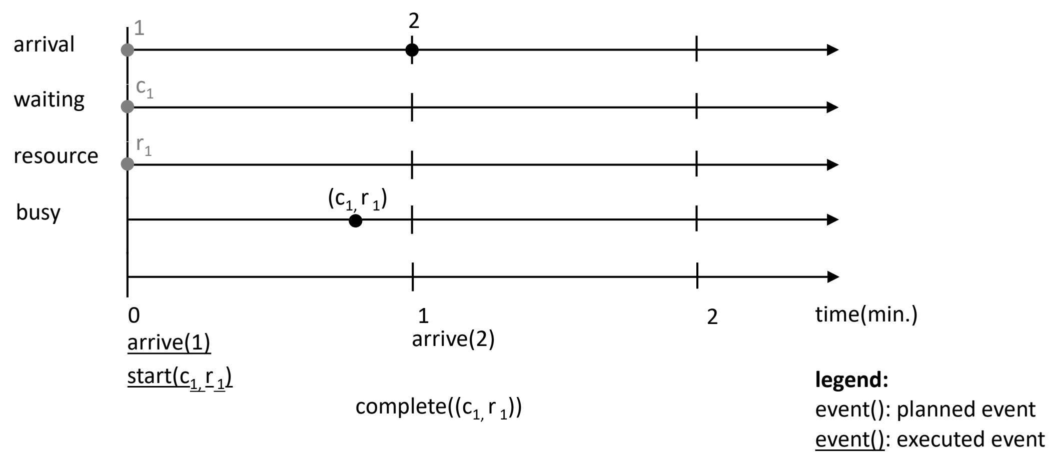

Figure 1.3: the example timeline with variables.

has started to be served. This timeline introduces the state variables `arrival`, `waiting`, `resource`, and `busy` and shows how events change the values of those variables and, consequently, of the state of the system. We use the arrival variable to represent the arrival of the next customer. At time t=0, the value for `arrival` is 1, meaning that at that time the first customer (with number 1) can arrive. Also, the value for resource is r_1, representing the available cashier. When the arrive event happens, the value for `arrival` is removed, which is why it is greyed out in the timeline. `arrival` produces a new value for arrival: the value 2 at time t=1. This is the moment at which the next customer can arrive. It also produces a value c_1 for the variable `waiting`, representing that customer c_1 has arrived and is available in the waiting variable. Next, the start event happens. It takes the values from `waiting` and `resource` and produces a value (c_1, r_1) in the `busy` variable, representing that resource r_1 is now busy serving c_1. The value for c_1 in the `waiting` variable and for r_1 in the `resource` variable are removed.

1.2 Modeling a Discrete Event Simulation

The timelines from the previous section show what has happened in a particular simulation scenario. However, to model a discrete event simulation in general, we need to model all the possible states of the system and how events change the state in general.

We model the possible states of the system by modeling the variables that represent that state. In our example, we introduced the variables `arrival`, `waiting`, `resource`, and `busy` as representing the state of the system.

We can then define conditions on when an event can happen as conditions on the state variables of the system. Moreover, we can define events as functions that take state variables as input and produce values for (other) state variables. For example, the condition for starting to serve a customer would be that there is a customer in the variable `waiting` and a cashier in the variable `resource`, and when we start to serve the customer, the customer and the cashier can be removed from their variables and put into another variable `busy`, which represents that the resource is busy serving a customer.

More precisely, we define simulation behavior rules in line with the theory of Petri nets as follows:

- an event can happen when each of its specified parameter variables has a value; and

- when an event happens, it removes a value from its specified parameter variables and puts a value in its return variables.

In this way, we do not explicitly have to model the constraint that there must be a customer waiting and a cashier resource available, because if `waiting` and `resource` are input parameters of the `start` event, this is an implicit requirement: the `start` event can only happen when each of its specified parameter variables, `waiting` and `resource`, have a value.

We can give variables a value with a delay, meaning that the values are not immediately available, but only when the simulation reaches a particular moment in time. This allows us to influence the moment in time at which events happen, because events can only happen when their parameter variables have a value. In this way we can model interarrival time, by only allowing the next arrival to happen after a specified interarrival time. We can do that by making the `arrive` event depend on a variable that we give a value with intervals determined by the interarrival time. Similarly, we can model processing time, by only allowing service to complete after the processing time has passed, i.e. by making the `complete` event depend on a variable that is available after the processing time has passed.

Finally, to enable the simulator to start, we need to set up an initial state of the system, from which the first events can start. For our example system that would be a state in which no customers have arrived, but there is a single resource available. We then define the `start` event as follows in pseudo code.

```
def start(w: waiting, r: resource): busy
  return (w, r)@time+exp(1/0.8)
```

This definition states that the start event uses a value from the `waiting` variable and the `resource` variable, which it will internally refer to as `w` and `r`. It returns a value for the `busy` variable. The function body specifies that this value is produced as the tuple that is the combination of the input values `w` from the `waiting` variable and `r` from the `resource` variable. It produces this value with a delay, represented by the @ notation. The delay compared to the current time is a draw from an exponential distribution with a rate $\lambda = \frac{1}{0.8}$, which is equal to the service time of the example. In other words, when busy, the customer and the resource are available again after on average 0.8 minutes. Figure 1.4 graphically represents the event as a Petri net. In this representation, the variables are represented as circles - called places in Petri net theory - and the events are represented as squares - called transitions in Petri net theory. The Petri net represents the same as what was just introduced: that the start event takes values from resource (calling it *r*) and from waiting (calling it *w*) and produces a value in busy according to the formula that is provided.

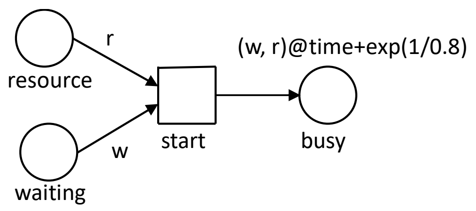

Figure 1.4: Petri net representation of the start event.

We define the `complete` event as follows in pseudo code.

```
def complete(b: busy): resource
  return b[1]@time
```

Note that, because of the delay that the value of busy gets from the start event, the complete event can only happen after the processing time has passed. When it happens, it puts the value of the cashier - the second element of the busy tuple b - back into the resource variable. It does so with immediate effect, meaning that the cashier resource is available immediately upon completion of the service. Figure 1.5 shows a Petri net representation of the complete event.

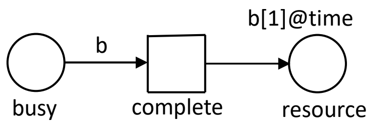

Figure 1.5: Petri net representation of the complete event.

We define the arrive event as follows.

```
def arrive(a: arrival): arrival, waiting
  return (a+1)@time+exp(1), 'c'+str(a)@time
```

This event can only happen when there is a value in the arrival variable. We can enable this by putting the value 1 at time $t = 0$ into the arrival variable at the initialization of the simulation. We will use this value to represent the unique identifier of the next customer to arrive. By putting this value into the arrival variable the arrive event can happen right at the start of the simulation. When it happens, it produces values in the arrival and the waiting variables. The arrival value is produced with a delay according to the interarrival time, such that the next time the arrive event can happen is after the interarrival time. The value that it produces in the waiting variable is a string representing the customer identifier. This value is produced without delay, meaning that after arrival, the customer is waiting immediately. Figure 1.6 shows a Petri net representation of the simulation scenario as a whole, integrating the three events that are defined above. The benefit of this representation is that it is possible to get an overview of how the simulation model and the various events and variables fit together as a whole.

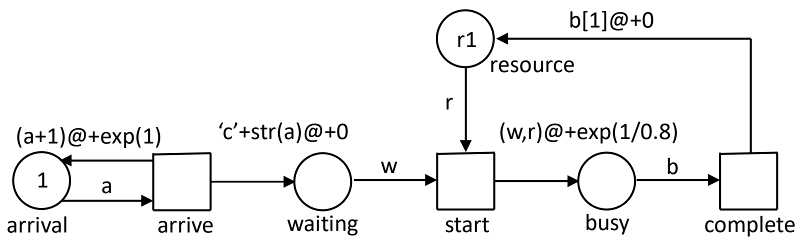

Figure 1.6: Petri net representation of the simulation scenario.

1.3 Modeling a Discrete Event Simulation in SimPN

The theory of Discrete Event Simulation is implemented in a Python library called SimPN. There exist many dedicated business process simulation tools. This includes simulation tools that have specifically been designed to simulate customer-oriented business processes modeled in the BPMN modeling language, such as Signavio. It also includes general-purpose simulation tools, such as Arena, that can both be used to model customer-oriented business processes and resource-oriented business processes, such as transportation processes or production processes. There even exist general purpose tools, such as CPN Tools, that are based on Petri nets. The benefit of using such tools is that simple simulations are relatively easy to build and reporting facilities are built-in, including the possibilities for creating great-looking graphs and consultancy reports. The main benefit of using SimPN is the flexibility that you have, because it is directly integrated into your programming environment. Due to that flexibility, you can, for example, directly connect your simulation to the database of your currently active cases and occupation of your resources. You can then do a simulation to see if your currently active cases are likely to complete on time and take action if the simulation tells you that they will not. Such a model of your process as it is currently running is also known as a digital twin. SimPN is ideally suited for creating a digital twin of your process, because it can not only easily be connected to the real-time data of what is going on in your process, but also to prediction and optimization libraries that can be used to calculate the appropriate action in the current state of your process. Because of that flexibility, SimPN is a great tool to have in your programming toolkit. There exists another good library for simulation in Python, called SimPy. This library uses more specific simulation concepts, based on queuing theory. This makes SimPy easier to get started with, but also less expressive.

SimPN follows the modeling paradigm introduced in the previous section, in that to create a simulation model, you need to model:

1. the variables that describe the statespace of the system;
2. the initial state of the system; and
3. the events that describe when the system can change and what happens when it does.

This section explains how these three elements can be modeled in SimPN.

We must define the simulation model, which can simply be done by instantiating the `SimProblem` class as follows.

```python
from simpn.simulator import SimProblem

shop = SimProblem()
```

This code imports the relevant library elements and creates a new simulation model in a variable named `shop`.

In our example, the state of the system consists of a variable that indicates the sequence number of the next customer to arrive, customers that are waiting in line at the cash register, resources that are free to help the customer, and resources that are busy helping a customer. Consequently, we can model the state of our simulation by instantiating and naming them as follows.

```python
arrival = shop.add_var("arrival")
waiting = shop.add_var("waiting")
resource = shop.add_var("resource")
busy = shop.add_var("busy")
```

Having defined our variables, we can now model our initial state by assigning values to the variables as follows.

```python
resource.put("r1")
arrival.put(1)
```

A simulation variable is different from a regular Python variable in two important ways. First, a simulation variable can contain multiple values, while a regular Python variable can only contain one value. This is why we use `put` on a variable rather than an assignment. `put` allows us to put multiple values into a variable. For example, we could add another `resource.put("r2")` statement into the code, which would add another cashier resource. Second, values of a simulation variable are available from a specific moment in (simulation) time. By default, values that are put into a variable are available immediately from the start of the simulation time 0. However, we can change that by adding a time of availability to put statement. For example, changing the arrival to `arrival.put(1, time=1)` would make the value available at simulation time 1.

Finally, we can define our simulation events. The events are defined similarly to the pseudocode in the previous section. However, they are defined in two parts. The start event is defined as follows.

```python
from simpn.simulator import SimToken
from random import expovariate as exp

def start(c, r):
  return [SimToken((c, r), delay=exp(1/0.8))]

shop.add_event([waiting, resource], [busy], start)
```

First, let's focus on the `shop.add_event` statement. This tells the simulator that our event takes a value from the `waiting` variable and a value from the `resource` variable as input, produces a value for the `busy` variable as output, and uses the `start` function to do so.

The `start` function defines how the event modifies the system state (variables). Taking a value from the `waiting` variable (and calling it c) and a value from the `resource` variable (and calling it r), the function returns a tuple (c,r), representing that customer c is being processed by resource r. This return value will be put into the `busy` variable, as per the `shop.add_event` definition. However, as you can see, there are several things going on in the return statement.

First, the function does not return a single value, but a list of values. This is simply a convention that you have to remember: event functions return a list of values. The reason for this is that we defined the simulation event in `shop.add_event` as taking a list of values (consisting of one value from `waiting` and one value from `resource`) as input and as producing a list of values (consisting of one value for `busy`) as output. Accordingly, we must produce a list of values as output, even if there is only one value.

Second, the function does not return the tuple (c,r) itself, but returns a `SimToken` containing the resource. That is because in simulation, values have a time from which they are available. A value with a time is called a *token*. This represents that the value is only available at, or after, the specified time. In this case, the resource value is made available after a delay that is exponentially distributed with a $\lambda = \frac{1}{0.8}$.

Similarly, we can define the `complete` and `arrive` events as follows.

```python
def complete(b):
  return [SimToken(b[1])]

shop.add_event([busy], [resource], complete)

def arrive(a):
  return [SimToken(a+1, delay=exp(1)), SimToken('c' + str(a))]

shop.add_event([arrival], [arrival, waiting], arrive)
```

Now we have modeled the entire system and we can simulate it. To do that, we call the `simulate` function on the model. This function takes two parameters. One is the amount of time for which the simulation will be run. The other is the reporter that will be used to report the results of the simulation. In our example we will run the simulation for 2.5. We will use a `SimpleReporter` from the reporters package to report the result. This reporter simply prints each event that happens to the standard output.

```python
from simpn.reporters import SimpleReporter

shop.simulate(2.5, SimpleReporter())
```

Running the code should output something like the following, representing that the first customer arrives at time 0, is started immediately and completes at time 0.8 and so on.

```
arrive{arrival: 1}@t=0
start{waiting: c1, resource: r1}@t=0
complete{busy: ('c1', 'r1')}@t=0.8
arrive{arrival: 2}@t=1
start{waiting: c2, resource: r1}@t=1
arrive{arrival: 3}@t=1.7
complete{busy: ('c2', 'r1')}@t=1.9
start{waiting: c3, resource: r1}@t=1.9
```

1.4 Guard Conditions

Events can only happen when the variables that they use have a value. However, in many cases we want events to only happen in case variables have a specific value. For example, we only want breakable items in a warehouse to be packaged or we only want to accept customers for a loan who have a high enough credit rating. To place such specific constraints on when an event can happen, we can add a *guard condition* on the event.

A guard condition specifies the precise values for which an event can happen. For example, the condition that only customers with a credit rating over 100 can get a loan can be expressed as follows.

```python
customers = shop.add_var("customers")
customers.put(("c1", 90))
customers.put(("c2", 110))

```

```
5  accepted_customers = shop.add_var("accepted customers")
6
7  def accept(c):
8    return [SimToken(c)]
9
10 def acceptance_condition(c):
11     return c[1] > 100
12
13 shop.add_event([customers], [accepted_customers],
14                accept, guard=acceptance_condition)
```

This code has an event that takes customers from the `customers` variable and puts them into the `accepted_customers` variable (line 13-14). However, as you can see this event has a guard condition (line 10-11). The guard condition is a function that evaluates to true if the second element of the customer tuple, which is where we store the credit rating of the customer (see line 2-3), is greater than 100. Consequently, this event can only happen for customers that have a credit rating greater than 100.

1.5 Manipulating a Complete Queue

Note that all event behaviors that we have defined so far work on a single value in a variable. However, in simulation it is often necessary to manipulate all values at the same time. This is especially true when the values represent items in a queue, such as customers. A simple example that requires queue manipulation is the situation in which customers will leave if they see a long queue ahead of them, because this requires that we keep track of the number of customers in the queue. Another example that requires queue manipulation is the situation in which some customers leave the queue when they have been waiting for a long time, while others decide to stay even if they have been waiting long.

While it is possible to implement these examples using the original definition of Petri nets, this can potentially complicate the model. For example, it is possible to maintain an additional variable `nr_of_customers` to keep track of the number of customers in the queue and increase and decrease this variable when a customer enters or leaves the queue. However, this requires us to define this variable and make sure that its value is kept up do date, while in principle this could be as simple as checking the number of tokens in a variable. This can become cumbersome if we have to do that a lot. Therefore, we will allow for a slight abuse of the Petri net notation and support handling variables as queues.

Figure 1.7 illustrates for the running example how we can treat a variable as a queue. The key to doing this is not getting a single value `w` from the `waiting` variable, but getting the entire waiting queue. We can do this by referring to `waiting.queue`. In the figure below we then introduce a prototype `ws` for `waiting.queue`, but this is purely for readability purposes.

After we obtained the queue `ws`, we can manipulate it. The queue is obtained as a list and we can perform normal list operations on it, such as getting a specific element from the list or getting the length of the list. For the example we will replicate the normal behavior of the start event: it gets the first customer from the queue and puts it, together with a resource, in the busy place with a delay. However, now that we can manipulate entire queues, more complex behavior is possible as well.

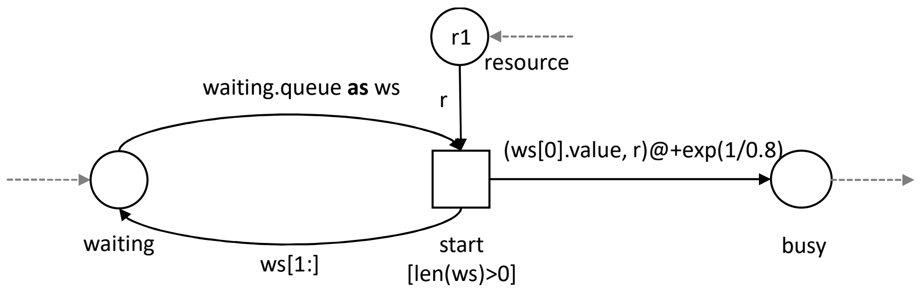

Figure 1.7: Treating a variable as a queue.

It must be noted that the queue is obtained from the variable as a token. This has two important implications. First, the queue has a time. However, the time of a queue token is always 0, meaning that a queue is always available. Second, the queue will be removed from the variable. This means that it also has to be put back, otherwise not just the first customer will be removed from the queue, but all customers will be removed from the queue. For that reason, now you also see an arrow back to the waiting variable that puts back all elements from the queue except the first element. Also note that the elements in the queue are tokens themselves. This means that they are stored in the queue with both their value and their time. Consequently, if we are only interested in the value of a token we have to explicitly refer to that `value` using the dot notation. Similarly, we can obtain the `time` of a token using the dot notation. This may be useful, for example, if we want to check how long a customer has been waiting in the queue. Finally, it is important to see that the queue can also be empty, in which case we cannot get a customer from the queue. However, when the queue is empty, there is still a queue - en empty queue - to allow the start event to occur. This will lead to an error, because we cannot get a customer from an empty queue. Consequently, we have to prevent the start event from happening when the queue is empty. The guard condition on the start event takes care of that.

In Python we can represent this behavior as follows. This is indeed far more complex than the definition above, but on the bright side, we can now manipulate queues, which will allow us to create more complex models in a simpler way further on in this chapter.

```python
def start(ws, r):
  c = ws[0].value
  return [ws[1:], SimToken((c, r), delay=exp(1/0.8))]

def start_condition(ws, r):
  return len(ws) > 0

shop.add_event([waiting.queue, resource], [waiting.queue, busy],
               start, guard=start_condition)
```

1.6 Simplifying the Simulation Model

There exist commonly occurring concepts that require us to write the same code structure very frequently. For example, as every task has a start and a complete event, we always need to model both for each task. This can become rather cumbersome. For that reason we introduce 'prototypes'. A prototype is a relatively short statement that expands into a more complex structure. Using prototypes we can represent commonly occurring constructs in a compact manner. For example, we can represent the task construct, which consists of multiple events and an intermediate 'busy' variable, in a more compact manner using the BPMNTask prototype as follows.

```
def start_scan(c, r):
  return [SimToken((c, r), delay=exp(1/9))]
BPMNTask(shop, [waiting, cassier], [done, cassier],
         "scan", start_scan)
```

While prototypes can be used for shorter representation, they explode into the common constructs that they represent. This means that they do not change the simulation model itself. Figure 1.8 illustrates this for the BPMNTask prototype. The solid black parts of this model represent what is specified in code above. The dashed grey parts represent what the prototype expands into in the simulation model. So while the user just specifies a task labeled 'scan' with a particular start_scan function, this expands into a start scan and a complete scan event as usual, with a variable in between that represents that the resource is busy with the customer. It is just not necessary to specify these events and the variable separately anymore.

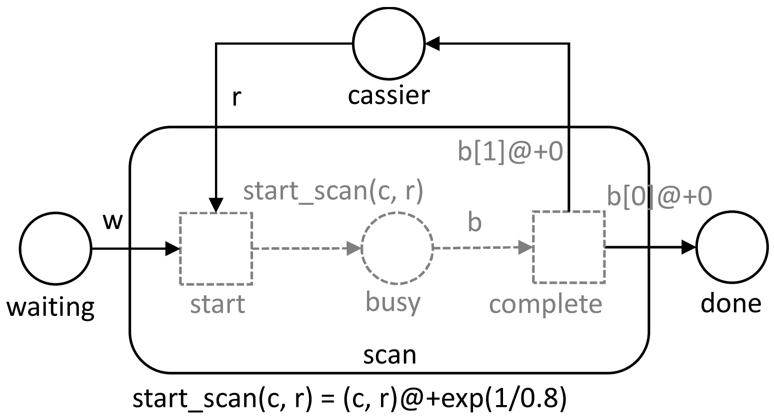

Figure 1.8: expansion of the task prototype.

Lambda functions make it possible to further simplify the model, again not changing anything about how it works. A lambda function is a function that does not have a name and only has a return statement. A normal function that only has a return statement can always be rewritten as a lambda function. This will again make the code more compact. For example, consider the function plus that adds up two numbers.

```
def plus(a, b):
    return a + b
```

This function can be rewritten as a lambda function `lambda a, b: a + b`. Note that this lambda-function does not have a name and only contains a return statement (without actually calling it `return`). Using prototypes and lambda functions, we can rewrite the simulation model of the shop as follows.

```python
shop = SimProblem()

waiting = shop.add_var("waiting")
done = shop.add_var("done")
resource = shop.add_var("resource")
resource.put("r1")

BPMNStartEvent(shop, [], [waiting], "arrive", lambda: exp(1))

BPMNTask(shop, [waiting, resource], [done, resource],
         "scan_groceries", lambda c, r: [SimToken((c, r),
         delay=exp(1/0.8))])
```

While this code is much shorter than the original code, it expands into roughly the same simulation model as before. The main difference is that an additional `done` variable is added where customers are stored after they are done.

1.7 Reporting Measurements

SimPN allows you to monitor each event that happens in the simulator. You can use that information to generate reports. There are some standard reporting functions, but you can also create your own.

This section first explains how you can generate the most common measurements. It then explains how you can store detailed results per customer case, which can be used to calculate advanced measurements at a later stage. Finally, it explains that, when simulating, you have to take warmup times and replications into account in order to produce valid measurements.

1.7.1 Simple Reporting

To monitor the events that happen in the simulation, you need to create a reporter. A reporter is a subclass of the `Reporter` class that implements the `callback` function. If you give your reporter to the simulator, the simulator invokes the callback function each time an event happens, allowing you to do reporting on the events as they happen. The following code represents the simplest reporter that you can make. It simply prints each event that happens.

```python
from simpn.reporters import Reporter

class MyReporter(Reporter):
    def callback(self, timed_binding):
        print(timed_binding)

shop.simulate(3.0, MyReporter())
```

This reporter outputs information that looks something like this:

```
([(arrival, 1@0)], 0, arrive)
```

```
([(waiting, c1@0), (resource, r1@0)], 0, start)
([(busy, ('c1', 'r1')@0.8)], 0.8, complete)
([(arrival, 2@1)], 1, arrive)
([(waiting, c2@1), (resource, r1@0.8)], 1, start)
([(arrival, 3@1.7)], 1.7, arrive)
([(busy, ('c2', 'r1')@1.9)], 1.9, complete)
([(waiting, c3@1.7), (resource, r1@1.9)], 1.9, start)
([(busy, ('c3', 'r1')@2.6)], 2.6, complete)
```

This shows the information that is provided with each event, i.e. the `timed_binding` with which the callback function is invoked. Each `timed_binding` has three parts:

1. the list of (timed) variable values that triggered the event;
2. the time at which the event happened; and
3. the event that happened.

The timed variable values are provided as a list of tuples, where the first element is the variable name and the second element is the timed value (the token) of that variable. For example, the first event above happened with the value 1 for the `arrival` variable, which was available at time 0. The event that happened was `arrive` and it happened at time 0. In other words: at time 0 the first customer arrived.

We can use this information to create more interesting measurements and reports. Typically, there are four measurements that are of interest:

- the average *processing time* per customer, which is the time during which tasks are performed with customers;
- the average *waiting time* per customer, which is the time a customer spends waiting, usually for a resource to become available;
- the average *sojourn time* or *cycle time* per customer, which is the total time between the moment the customer arrives and the moment the customer is ready and leaves; and
- the *utilization rate* of a resource, which is the fraction of the time that resource is busy.

In the example at the end of the previous section, the simulation took 2.5 minutes in total. The first customer spent 0.8 minutes in processing, the second 0.9 minutes (the difference between starting to process the customer at time 1 and completing it at time 1.9), and the third 0.7 minutes, such that the average processing time was 0.7 minutes. The first customer spent 0 minutes waiting, the second also spent 0 minutes waiting (the difference between starting to process the customer at time 1 and the customer's arrival at time 1). The third customer spent 0.2 minutes waiting, such that the average waiting time rounded to two decimals was 0.07 minutes. The first customer arrived at $t = 0$ and was ready at $t = 0.8$, the second customer arrived at $t = 1$ and was ready at $t = 1.9$, and the third arrived at $t = 1.7$ and was ready at $t = 2.6$. Consequently, the average sojourn time rounded to two decimals was 0.87 minutes, the same as the sum of the average processing time and waiting time.

The cassier worked between $t = 0$ and $t = 0.8$ on customer 1, between $t = 1$ and $t = 1.9$ on customer 2, and between $t = 1.9$ and $t = 2.6$ on customer 3. Consequently it was busy for $0.8 + 0.9 + 0.7$ over 3.0 or 80% of the time.

We can keep track of the measurements that are of interest by keeping them as variables of the reporter class as follows.

```python
class TimesReporter(Reporter):

  def __init__(self):
    self.arrival_times = dict()
    self.start_times = dict()
    self.complete_times = dict()
    self.total_wait_time = 0
    self.total_proc_time = 0

  def callback(self, timed_binding):
    (binding, time, event) = timed_binding
    if event.get_id() == "arrive":
        c_id = binding[0][1].value
        self.arrival_times[c_id] = time
    elif event.get_id() == "start":
        c_id = binding[0][1].value
        self.start_times[c_id] = time
        self.total_wait_time += time - self.arrival_times[c_id]
    elif event.get_id() == "complete":
        c_id = binding[0][1].value[0]
        self.complete_times[c_id] = time
        self.total_proc_time += time - self.start_times[c_id]
```

To make this code work, we slightly changed the `arrive` function. Instead of creating an identifier of each customer that is a string containing “c” followed by a sequence number, we changed it such that each customer identifier is an integer sequence number. In this way we can uniformly track the customer through the process by number, because the `arrive` function itself also only receives a number as input.

The code above stores the arrival time, the start time, and the completion time of each customer. It also stores the sum of the waiting times of all customers and the sum of the processing times of all customers. This is all done in the `callback` function. The `callback` function splits up the event information into the binding of variables to values, the time at which the event happened and the event that happened. When a customer arrives, it gets the customer identifier from the variable bindings. The customer identifier is obtained by taking the first element of the binding (`binding[0]`), which is a pair of a variable and a token. Then taking the token in that binding (`binding[0][1]`) and taking the value of the token (`binding[0][1].value`). Subsequently, we store the customer’s arrival time in the `arrival_times` dictionary. When processing a customer starts we do something similar to store the customer’s start time and we also add the time the customer has waited (which is the difference between the moment at which the customer arrived and the current time) to the total waiting time. Storing the processing time works similarly.

With this code, once the simulation has completed, the sum of all waiting times and the sum of all processing times is stored in the corresponding variables. We can then use that information to calculate the average waiting time, the average processing time and the average cycle time as follows.

```
class TimesReporter(Reporter):

  def mean_waiting_time(self):
    return self.total_wait_time / len(self.start_times)

  def mean_processing_time(self):
    return self.total_proc_time / len(self.complete_times)

  def mean_cycle_time(self):
    return self.mean_waiting_time() + self.mean_processing_time()
```

Clearly, having to write this code for each process is a hassle. For that reason, some standard reporters are provided, such as the `ProcessReporter`. This reporter reports on the typical process measurements mentioned above: the average waiting time per case, the average processing time per case, the average cycle time per case, and utilization rate of each resource. This reporter heavily relies on the use of process prototypes, i.e. it reports on the cycle time between `BPMNStartEvent` and `BPMNEndEvent` prototypes and it reports on the waiting times and processing times based on `BPMNTask` prototypes. If the simulation problem is specified with those prototypes, the reporter can generate the measures. The code below illustrates how this can be done for our running example.

```
BPMNStartEvent(shop, [], [waiting], "arrive", lambda: exp(1/10))

BPMNTask(shop, [waiting, cassier], [done, cassier],
         "scan_groceries",
         lambda c, r: [SimToken((c, r), delay=exp(1/9))])

BPMNEndEvent(shop, [done], [], "complete")

reporter = ProcessReporter()
shop.simulate(24*60, reporter)
reporter.print_result()
```

Another standard reporter is the `EventLogReporter`. This reporter stores all events that happen in a tabular format in a csv file. The information in the csv file can subsequently be analyzed to provide more detailed insight into the process. The code below shows how this reporter can be used.

```
reporter = EventLogReporter("reporting.csv")
shop.simulate(24*60, reporter)
reporter.close()
```

A great benefit of this way of reporting is that the result can easily be imported into a process mining tool where different techniques exist for further analysis. Figure 1.9 shows the result of mining information from the csv file in the process mining tool Disco[1]. The figure shows that during the simulation run 146 customers arrive. For 134 or them the scan groceries event started and it took on average 16.8 minutes between the arrival of the customer and the moment at which the scanning of the groceries started. For 12 customers, the scan groceries event never started. These are customers that were still waiting in queue when the simulation ended. The scanning of the groceries took on average 7.9 minutes, which is lower than expected from the problem specification, where scanning is specified to take on average 9 minutes, but this difference can be attributed to randomness.

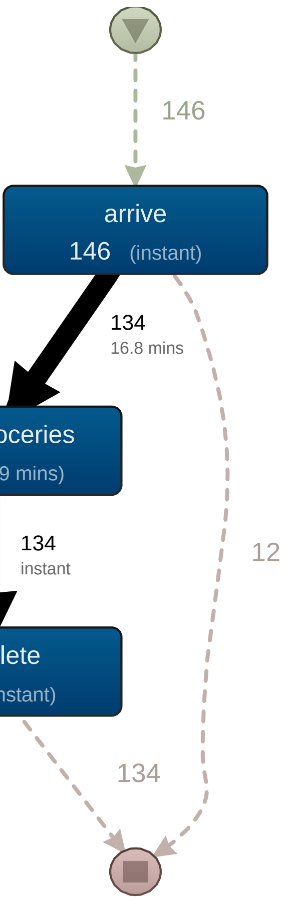

Figure 1.9: mined simulation log.

1.7.2 Warmup Time and Replications

To report valid measurements, two aspects must be considered: warmup time and replications.

[1] https://fluxicon.com/disco/

Warmup time must be considered, because when a simulation starts, the simulation model is empty, meaning there are no tasks being performed, all resources are available, and there are no waiting lines. For that reason the first case to arrive always sees an empty system in front of it and will have 0 waiting time. The second customer will see at most one other case in front of it and have a low waiting time. This will continue until the simulation model reaches a steady state: a state from which long-term averages of the simulation variables are not expected to change substantially anymore. This is a very relative statement, because - due to randomness - the variables will actually continue to change over time. The question is if they change substantially and that is usually a subjective decision. Figure 1.10 illustrates that best and usually, you will want to plot a graph like the one shown in this figure to determine the warmup time for a simulation. This graph is created using the standard `WarmupReporter`.

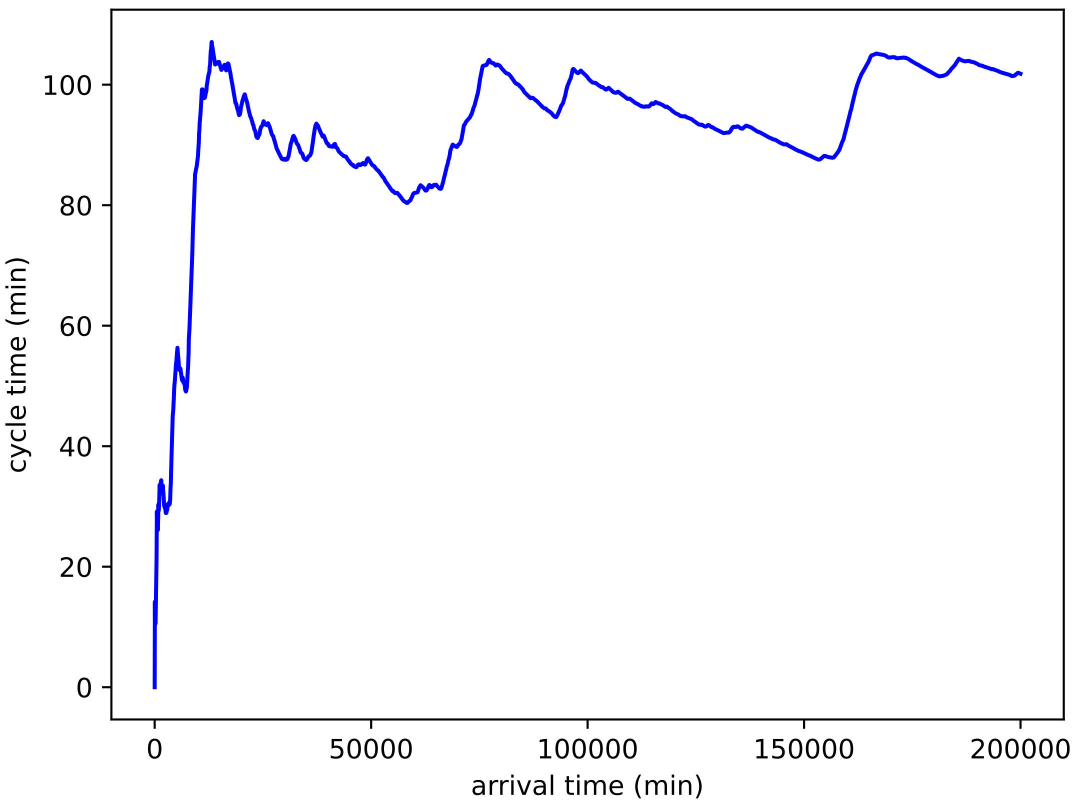

Figure 1.10: A graph that illustrates warmup time.

The figure shows the warmup time for the simulation of our running example. Specifically, it plots the average cycle time of all cases that have arrived until simulation time t. It clearly shows that initially the cycle time is very low and then rapidly increases over the first 25,000 minutes or so, before reaching an steady state. It subsequently continues to oscillate around the steady state, but that can be considered normal behavior.

Clearly, if we used the measurements of the average waiting time in the first 5,000 minutes, we would get an average that is (much) lower than in the steady state and therefore invalid. For that reason, we explicitly exclude events during the warmup time from our reported measurements. It is best to take a margin here, to ensure that we definitely do not include measurements that are still done in the warmup time. For our example, we could assume the warmup time to be 50,000 minutes and only report the measurements on events that happened after 50,000 minutes in simulation time.

It is recommended to also use warmup times in the situation of a supermarket, where it is actually valid that the system starts empty in the morning when the supermarket opens. In such simulations, it is better to identify different scenarios. A scenario is a variation of a simulation model in which the effect of specific settings are explored. In our example we could consider the scenario in which the supermarket has just opened and is filling up with customers separately from the scenario in which the supermarket is operating in its steady state during the day. For both of these scenarios we can have a different arrival rate and, optionally, a different number of resources. The problem with simulating a single scenario that includes both the interval in the morning and during the day is that you may end up dimensioning your supermarket for the 'average' case, meaning it will likely be overdimensioned for the morning and underdimensioned for the day interval and for that reason represent the worst of both worlds.

Replications are the second aspect of simulation that must be considered to produce valid measurements. They are repetitions of the simulation. Replications of a simulation are absolutely necessary, because simulations involve a lot of randomness, and we want to make sure that a conclusion that we draw is valid and not due to randomness.

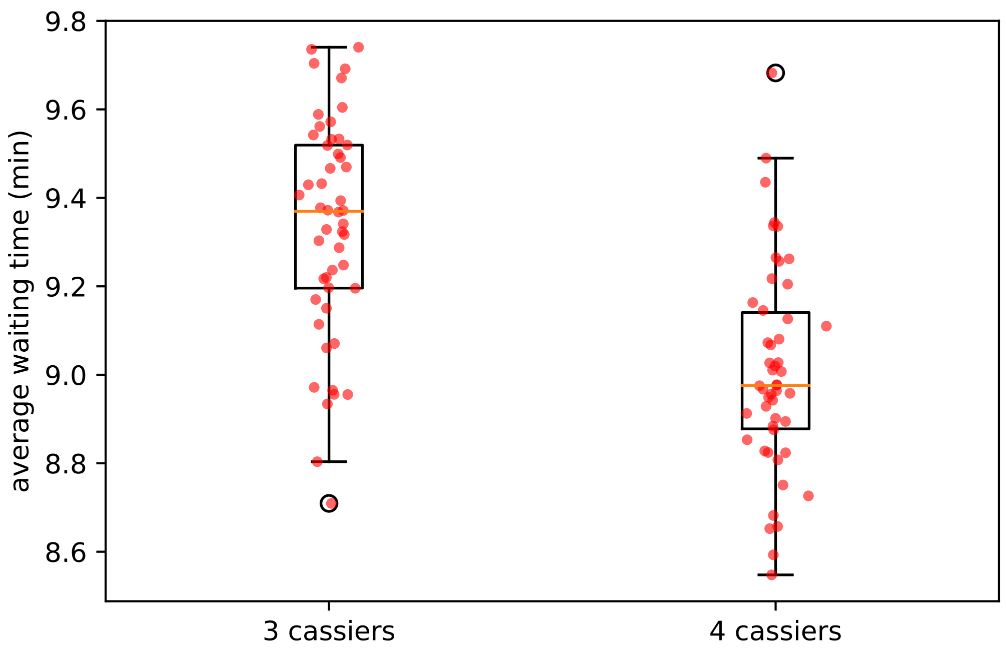

Figure 1.11: A graph that illustrates replications.

Figure 1.11 illustrates the need to do replications. The figure shows two scenarios for our running example. One in which there are 3 cassiers and one in which there are 4 cassiers. For both scenarios we measure the average waiting time of the customers and for both scenarios we did 50 replications. The small dots represent the average waiting time for a particular replication. To be precise: a warmup time of 20,000 minutes is considered and events that happen within the warmup time are not taken into account when computing the average waiting time. The average waiting time is computed over customers that arrive and start in the interval between 20,000 and 100,000 minutes of simulation time. Now suppose that we would do no replications for the scenarios. Then it is quite possible that we get a relatively low waiting time from the simulation with 3 cassiers (the lowest point on the graph has an average waiting time of around 8.7 minutes) and a relatively high waiting time from the simulation with 4 cassiers (the highest point on the graph has an average waiting time of around 9.7 minutes). This may lead us to conclude that having 3 cassiers leads to a lower waiting time for customers than having 4 cassiers, which is of course not true. It may of course be true on some days, when it happens that with 4 cassiers a lot of customers arrived with many groceries, while with 3 cassiers a lot of customers arrived with few groceries. However, this is a random effect and in general more cassiers means less waiting time for customers. This is indeed nicely illustrated by the box plots that are drawn for the replications. These illustrate that it is very likely that the average waiting time is higher for 3 cassiers than for 4. To be sure, you can still do a statistical test on the collections of replications.

The code below shows how you can do replications of a simulation and calculate results, while taking the warmup time into consideration. Key to this are the `store_checkpoint` and `restore_checkpoint` functions that can be used to reset the simulator to its initial state and run a new, fresh, simulation.

```python
shop.store_checkpoint("initial state")

average_cycle_times = []

NR_REPLICATIONS = 50
SIMULATION_DURATION = 40000
WARMUP_TIME = 20000
for _ in range(NR_REPLICATIONS):
  reporter = ProcessReporter(WARMUP_TIME)
  shop.restore_checkpoint("initial state")
  shop.simulate(SIMULATION_DURATION, reporter)

  average_cycle_times.append(reporter.total_cycle_time /
                             reporter.nr_completed)
```

1.8 Verification, Validation, and Sensitivity

Once you have built your simulation model, you should check if it is correct and if the conclusions that you draw from it are correct. There are three important checks that you can do for those purposes: verification, validation, and sensitivity analysis.

Verification of a simulation model concerns checking if the model behaves as you expect it to behave. When you constructed the model, you explicitly created a particular behavior regarding: the inter-arrival of customers, the tasks that are performed for these customers and the order in which they are performed, the amount of time these tasks take, and the resources involved in these tasks. When verifying the model you should check if the model behaves as you expect with respect to these aspects. There are two ways in which you can easily do that.

The first way to verify a simulation model is by adding print statements to your code that print each event that happens for a customer. You can use a `SimpleReporter` for that. You can then manually inspect if the customers behave in the way you expect them to behave. For ease of reading, you may want to start with one customer. You may want to run the simulation multiple times for one customer, to check multiple random behaviors from the simulation, such as if the different random draws from the processing time are reasonable, and - if there are alternative tasks begin performed - if the sequences of tasks are reasonable and if all sequences of tasks are indeed being performed. Next, you will want to try the same for multiple customers. This allows you to check if the sharing of resources between customers is indeed done correctly and if the inter-arrival times are indeed correct.

The second way to verify a simulation model is by applying process mining to the event log that can be generated by the simulator using the `EventLogReporter`. This allows you to produce a process model from the simulation like the one shown in Figure 1.9. This process model is very easy to inspect for correctness, especially if you constructed a process model before you created the simulation model. In that case you can do a direct comparison between the original process model and the mined process model.

Validation of a simulation model concerns checking if the model behaves similarly to the real-world process that you are trying to represent. The two main ways of validating your model are through face validation and through data validation.

Face validation involves asking one or more domain experts if they think the model behaves according to the process in the real world. Assuming you have already validated the process model on which you based your simulation, you should focus on measurements that are not immediately visible from the process model. Specifically, this includes: the waiting time, the cycle time, the resource utilization and the fraction of customer cases that have a particular final outcome. Face validation is a very imprecise way of validation, because domain experts may not know precisely what the values are for the parameters that are being checked. When using face validation, it is best to ask for quantitative evidence of the correctness of the measurements.

Data validation is a more precise way of validating if your model behaves according to the process in the real world. Similar to face validation, you will want to focus on validating measurements that are not immediately visible from the process model. Data validation concerns checking them against measurements made in the real world and assumes that such measurement data is available. For example, you can check the utilization rate calculated by your simulation model to the actual measured percentage of time the people who work in the process in the real world are actively working. The clear drawback of this form of validation is that such measurements may not always be available or accurate.

Sensitivity analysis concerns checking how sensitive particular measurements, calculated by the simulation model, are to changes in the parameters of the model. In this context we mainly use it to get an idea of the validity of our conclusions in light of imprecision of our parameters. For example, we may already know that resources are only available to work in the process around 70% of the time (we will see further on in this chapter how we can implement that in the simulation model) and we want to check if 99% of our customer cases are handled within 2 weeks. We may also know that the 70% availability of our resources is an imprecise estimate. In that case we will want to check how sensitive our conclusion, that 99% of the customer cases are handled within 2 weeks, is to the availability of the resources; does the conclusion still hold if the availability is 65% or 60%?

1.9 Simulating Business Process Constructs in SimPN

Above, we started by modeling a very simple process with one task. Of course multiple tasks can be performed for a customer case as well and these tasks can be performed in sequence, as alternatives to each other or in parallel. We will consider these and other constructs for adapting the structure of the business process in this section.

1.9.1 Sequential Tasks

The first extension is putting multiple tasks in a sequence. Figure 1.12 shows an example of two tasks in a sequence. The first task is the original task for scanning the groceries and the second task is bagging the groceries, which is performed by another type of resource: a bagger.

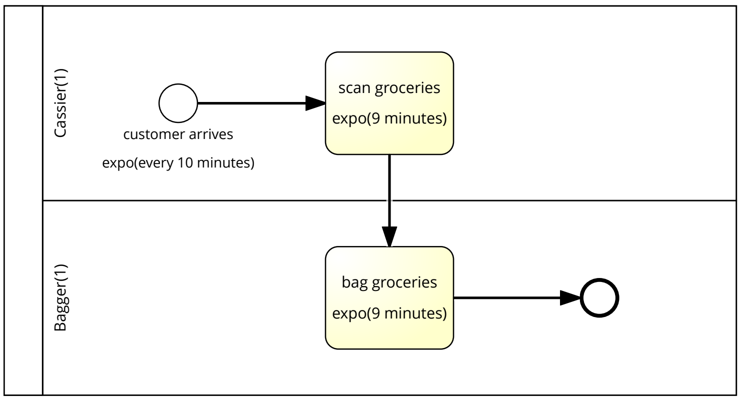

Figure 1.12: a process with sequential tasks.

A sequence of tasks can simply be implemented by repeating the code for starting and completing the next task. After completing the first task, the customer must be placed in the queue variable of the next task. The code below illustrates this. In this code, lines 3-5 and 13-15 show the original code for performing the scanning of the groceries, which is followed by similar lines of code 7-9 and 17-19 for performing the bagging of the groceries. As the bagging of the groceries involves a different resource, this resource also has to be created. Note that the complete event functions have been adapted to also pass the customer to a next variable. To that end, the `complete_scan` function returns a list with two tokens. The first token contains the resource from the tuple `b` and the second token contains the customer from the tuple `b`. These are placed in the `cassier` and `bag_queue` variables respectively.

```python
shop = SimProblem()

scan_q = shop.add_var("scan queue")
cassier = shop.add_var("cassier")
cassier.put("r1")

bag_q = shop.add_var("bag queue")
bagger = shop.add_var("bagger")
bagger.put("b1")

done = shop.add_var("done")

def scan(c, r):
  return [SimToken((c, r), delay=exp(1/9))]
BPMNTask(shop, [scan_q, cassier], [bag_q, cassier], "scan", scan)

def bag(c, r):
  return [SimToken((c, r), delay=exp(1/9))]
BPMNTask(shop, [bag_q, bagger], [done, bagger], "bag", bag)

def interarrival_time():
  return exp(1/10)
BPMNStartEvent(shop, [], [scan_q], "arrive", interarrival_time)

shop.simulate(60, SimpleReporter())
```

Clearly this code can easily be extended with any number of sequential tasks. It is also important to note that the second task in a sequence can also be performed by the same type of resource. To enable that, the start and complete events should simply use the same resource variable.

1.9.2 Choice

It is also possible to create alternative paths through a process as illustrated in Figure 1.13. In this figure, after performing the first task, there is a 25% chance that the customer will use the ATM. Accordingly, there is a 75% chance that the customer does not and just leaves.

Figure 1.14 illustrates this concept graphically. It is realized by the event putting a token in the variable of choice for further processing and no token (or `None`) in the other variable. This is realized by the following code, which randomly samples a value between 1 and 100 (a percentage). If that value is less than or equal to 25, it returns a token as its first output and no token as its second output, and otherwise the other way around. The first output is the queue for the ATM. The second output is the `done` variable, which represents customers that are done.

```python
def choose(c):
    percentage = uniform(1,100)
```

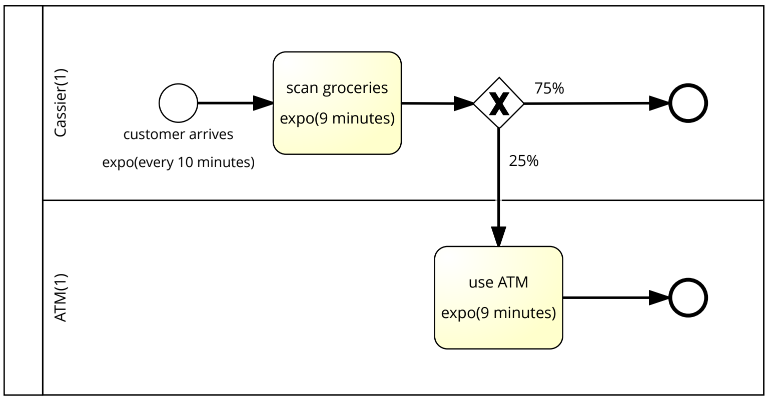

Figure 1.13: a process with choice.

```
    if percentage <= 25:
        return [SimToken(c), None]
    else:
        return [None, SimToken(c)]
shop.add_event([to_choose], [atm_queue, done], choose)
```

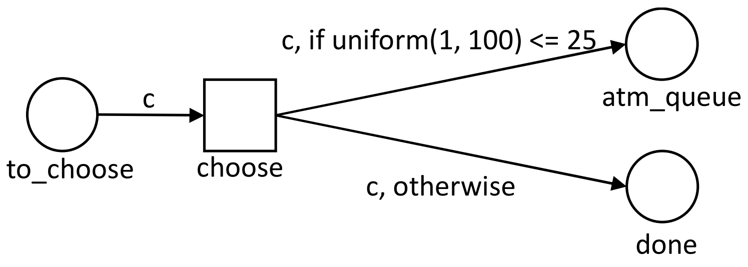

Figure 1.14: graphical representation of the choice construct.

1.9.3 Parallellism

It is of course also possible to perform tasks in parallel. Figure 1.15 shows this. In this figure, when the customer arrives the groceries are scanned and in parallel the ATM is used by the customer. The process only completes when both the task of scanning the groceries and the task of getting money from the ATM is done.

Figure 1.16 illustrates this concept graphically. We create this construct by adding two events, one to split the processing of the customer into multiple parallel paths and one to join the parallel paths again. The `split` event takes the customer from one variable and places copies of it into two variables. Now two tasks can happen at the same time, one using the copy from one variable and one using the copy from the other variable. This is because the rules state that "an event can happen when each of its specified parameter variables has a value". As there can be multiple events now with a customer value, multiple events can happen. The `join` event merges the parallel paths again by waiting for the same customer to arrive via both parallel paths, i.e. both its input variables. Note the guard condition that the values arriving must be the same (customer). Otherwise, it is also possible to join two parallel paths originating from different customers.

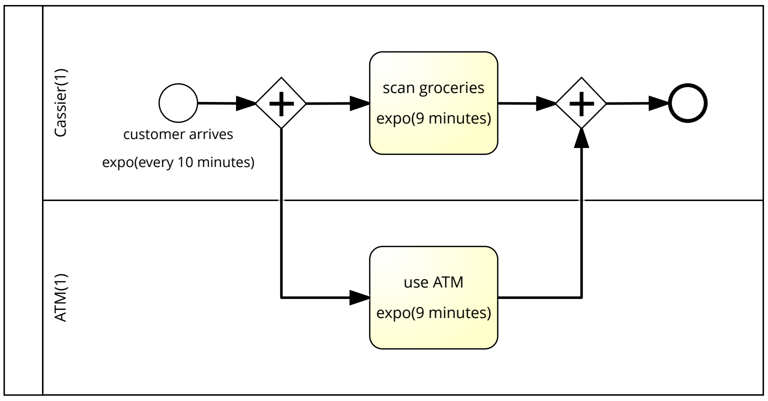

Figure 1.15: a process with parallellism.

```
shop.add_event([to_split], [scan_queue, atm_queue],
               lambda c: [SimToken(c), SimToken(c)], name="split")

...

shop.add_event([wait_sync_atm, wait_sync_scan], [to_done],
               lambda c1, c2: [SimToken(c1)], name="join",
               guard=lambda c1, c2: c1 == c2)
```

In more detail, the `split` event takes the token from the `to_split` variable and places copies of it into the `scan_queue` and `atm_queue` variables that are then input for the scan groceries and use atm tasks. The `join` event takes tokens from the `wait_sync_atm` and `wait_sync_scan` variables, where the customers are left after the respective tasks are completed. It checks if the tokens represent the same customer and then places one of the tokens (it does not matter which one, because they are the same anyway) in the `to_done` variable, from which the end event will be enabled.

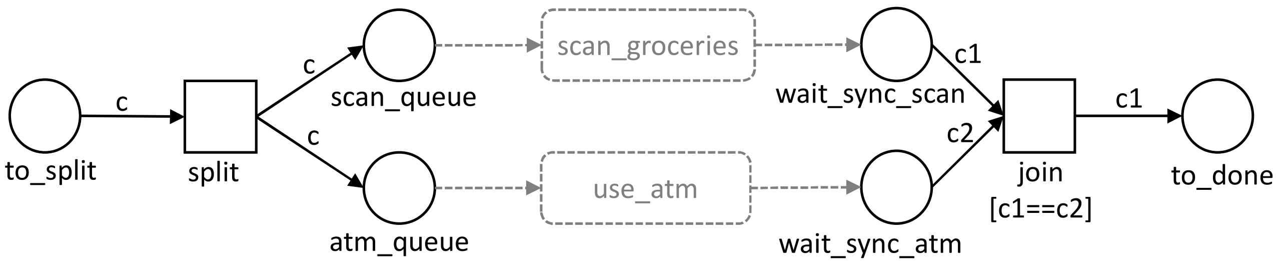

Figure 1.16: graphical representation of the parallel construct.

Clearly, the parallel construct, the choice construct and the sequence construct can easily be combined. For example, we can put the parallel construct in sequence with other tasks, like enter shop and leave shop. We can also put a task, such as bag groceries in parallel with the use ATM task, but in sequence with the scan groceries task. Similarly, we can combine these constructs with choice constructs to introduce choice in addition to sequential execution and parallelism of tasks.

1.9.4 Events

Figure 1.17 shows a process that contains an event. The process is different from the running example. It represents a simple sales process. In this process customer sales requests arrive at a rate that is exponentially distributed with an average of one per ten days. After a sales request arrives, first an offer is created, which takes a time that is exponentially distributed with an average of four days. The organization then waits for the response of the customer to the offer, which is done by means of an event. An event differs from a task in that it does not require the resources to do anything, i.e. the administrator is not needed for the event. It simply involved waiting until the event occurs. The resources can process tasks in the meantime. The waiting time for the event to occur is exponentially distributed with an average of four days. After that the response must be processed as a task performed again by the administrator.

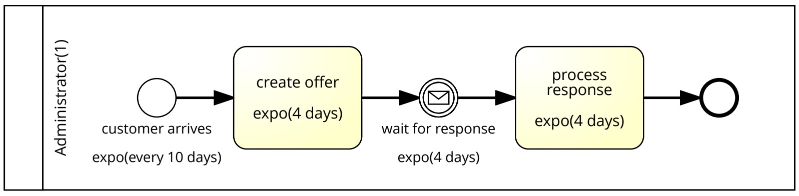

Figure 1.17: a process with events.

The code below shows how an event can be simulated in Python. It is relatively easy and - like a task - involves waiting for the required time, but - unlike a task - it is not necessary to use a resource.

```python
BPMNStartEvent(sales, [], [offer_queue],
               "customer_arrived", lambda: exp(1/10))

BPMNTask(sales, [offer_queue, administrator],
         [to_response, administrator], "create_offer",
         lambda c, r: [SimToken((c, r), delay=exp(1/4))])

sales.add_event([to_response], [processing_queue],
                lambda c: [SimToken(c, delay=exp(1/4))],
                name="wait_for_response")

BPMNTask(sales, [processing_queue, administrator],
         [done, administrator], "process_response",
         lambda c, r: [SimToken((c, r), delay=exp(1/4))])
```

1.9.5 Deferred Choice

A deferred choice is a choice that cannot be made immediately, but has to be made when some event happens at a future point in time. This is unlike a regular choice that can be made immediately at the moment the choice is encountered in the process. Figure 1.18 shows an example of a deferred choice. The choice has to be made after creating an offer. However, it cannot be made immediately. The process has to wait either until the customer sends the response to the offer, or until 5 days have passed, whichever happens first. The process can continue when either one of these events occurs.

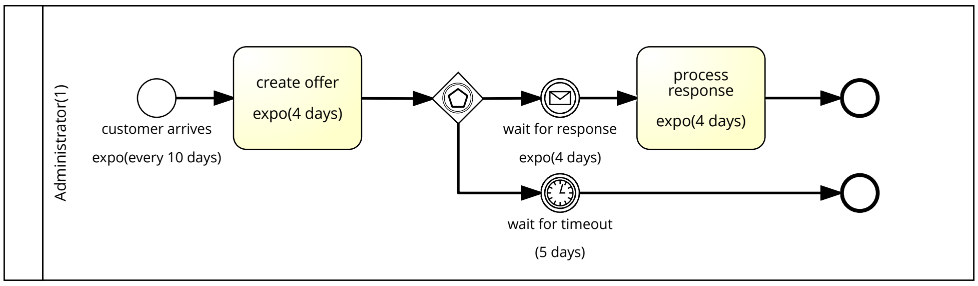

Figure 1.18: a process with deferred choice.

The code below shows how this process can be modeled in Python. Here, we mimic the deferred choice with a regular choice. The choice is made by sampling the time it takes the customer to respond. If this is less than 5 days, a token is generated with that waiting time. If the waiting time is 5 days or more, it is capped at 5 days. The reason for that is that the customer will be removed from the process in that case and consequently the wait will never be for more than 5 days.

We can use the waiting time information to determine what happens next. If waiting for a response takes less than 5 days, the process waits for the response by placing a token in the `awaiting_response` variable. Since the token has the sampled waiting time as delay, it will only be available after the sampled waiting time. Then the task is enabled to process the response. Otherwise, if the response takes 5 days or longer to arrive, the process waits for the timeout after 5 days by placing the token in the `to_leave` variable instead with 5 days as a delay.

```python
def choose(c):
  waiting_time = exp(1/4)
  if waiting_time < 5:
    return [SimToken(c, delay=waiting_time), None]
  else:
    return [None, SimToken(c, delay=5)]

shop.add_event([to_choose], [awaiting_response, to_leave], choose)
```

For this particular process it may not be necessary to wait for the timeout after 5 days, because nothing happens during or after that time. However, in general it is good to also handle the timeout event according to the principles of the simulation. Of course if we add reporting functionality after the timeout or some other task to remind the customer that they should still submit the response, it is necessary to wait for the timeout after 5 days.

In the example above we mimic the deferred choice with a regular choice. We can do that, because we can look into the future by already generating the time it takes to wait for a response before actually having to wait. However, it is not always possible to look into the future like that. If that is not possible, we cannot mimic the deferred choice with a regular choice. Figure 1.19 shows an example in which this is the case. This is an extension of the supermarket example, where the customer waits either for the cassier to become available or for a maximum of 15 minutes. If the 15 minutes expire before the cassier becomes available, the customer leaves before the task is performed. Note that if the task is started for the customer before the 15 minutes expire, the task will be completed for the customer, even if the 15 minutes expire while the task is being executed.

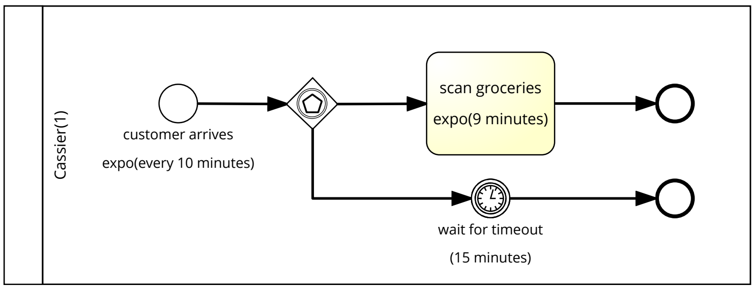

Figure 1.19: a process using deferred choice to maximize waiting time.

The code below shows how this deferred choice can be modeled. We introduce a deferred choice event that places the customer in the queue as usual, but also generated a token with a delay of 15 minutes, the timeout token. The leave event is enabled by this token and the customer in the queue. When the leave event is enabled (after 15 minutes), it can remove the timeout token and the customer from the queue. This must of course be the customer for which this particular timeout token was created and there is a guard condition that enforces this. The scan groceries task can also be performed as usual. After scanning the groceries an additional `remove` event is added. This event only exists to remove the timeout token. It takes the customer and passes it on as usual, but it also takes the timeout token. This event is not strictly necessary for the correct operation of the model, but it is good practice to remove any remaining tokens from the process before it completes. Figure 1.20 illustrates this construct.

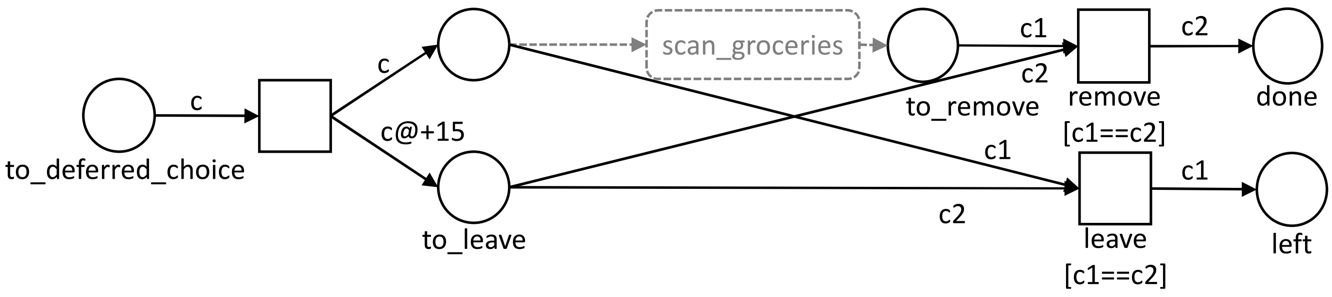

Figure 1.20: graphical representation of the deferred choice construct.

```
BPMNStartEvent(shop, [], [to_deferred_choice],
               "customer_arrived", lambda: exp(1/10))

shop.add_event([to_deferred_choice], [scan_queue, to_leave],
               lambda c: [SimToken(c), SimToken(c, delay=15)],
               "deferred_choice")
```

```

shop.add_event([scan_queue, to_leave], [to_left],
               lambda c, l: [SimToken(c)], "leave",
               guard=lambda c1, c2: c1==c2)

BPMNEndEvent(shop, [to_left], [], "left")

BPMNTask(shop, [scan_queue, cassier], [to_remove, cassier],
         "scan_groceries",
         lambda c, r: [SimToken((c, r), delay=exp(1/9))])

shop.add_event([to_remove, to_leave], [to_done],
               lambda c1, c2: [SimToken(c1)], "remove",
               guard=lambda c1, c2: c1==c2)

BPMNEndEvent(shop, [to_done], [], "done")
```

1.9.6 Different Case Types

It is possible that different types of customers arrive with different interarrival rates that have to be handled differently either in part or in full. Figure 1.21 shows an example of a process for which this holds. This is a variant of the sales process explained before. In this variant we see the arrival of customers with simple sales requests at an exponential rate with an average of one per 15 days. We see the arrival of customers with complex sales requests at an exponential rate with an average of one per 30 days. These types of customers behave similarly with respect to the creation of the sales offer and waiting for the response to that offer. However, their responses are processed differently. Most importantly, for the purposes of the simulation the processing of the different types of responses takes a different amount of time.

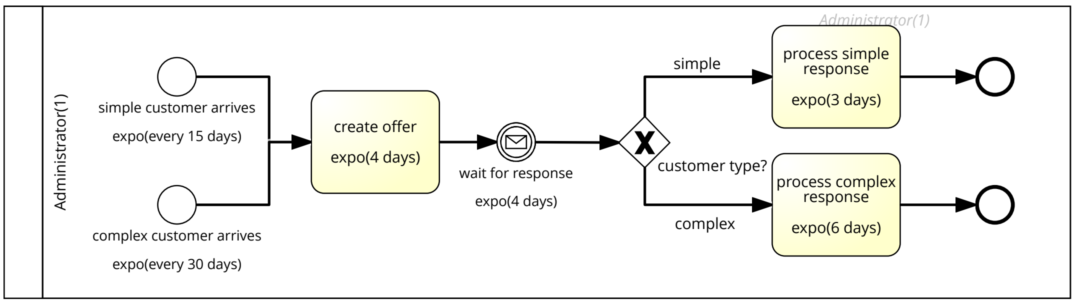

Figure 1.21: a process with different case types.

We create different types of customers with different arrival rates, using different start events as shown in the code below. The start events add a data field the tuples that represent the customer tokens, which carry either the value `simple` or `complex`. Note that the start event prototype automatically creates customer tokens as tuples, where the first element is the unique customer identifier, so each customer token looks like `(1, simple)`; a combination of a customer identifier and the data field. This property is used further on in the process to choose whether a simple response or a complex response is written to the customer.

```
BPMNStartEvent(shop, [], [offer_queue],
               "simple_customer_arrived", lambda: exp(1/15),
               lambda: [SimToken("simple")])
BPMNStartEvent(shop, [], [offer_queue],
               "complex_customer_arrived", lambda: exp(1/30),
               lambda: [SimToken("complex")])

BPMNTask(shop, [offer_queue, administrator],
         [to_response, administrator], "create_offer",
         lambda c, r: [SimToken((c, r), delay=exp(1/4))])

shop.add_event([to_response], [to_choose],
               lambda c: [SimToken(c, delay=exp(1/4))],
               name="wait_for_response")

def choose(c):
  if c[1] == "simple":
    return [SimToken(c), None]
  else:
    return [None, SimToken(c)]
shop.add_event([to_choose],
               [simple_response_q, complex_response_q], choose)

BPMNTask(shop, [simple_response_q, administrator],
         [to_done, administrator], "process_simple_response",
         lambda c,r: [SimToken((c, r), delay=exp(1/3))])
BPMNTask(shop, [complex_response_q, administrator],
         [to_done, administrator], "process_complex_response",
         lambda c,r : [SimToken((c, r), delay=exp(1/6))])

BPMNEndEvent(shop, [to_done], [], "done")
```

1.10 Simulating Various Resource Constructs

In the previous section we explained how a process can be extended by adding multiple tasks and various types of relations between those tasks. However, each of the tasks were performed by single resources that were always available. This can of course also be extended: there may be multiple resources, resources may work on a schedule - and consequently not always be available, and there may be different strategies for choosing which case to handle first. In this section we will consider how these - and other - constructs can be modelled.

1.10.1 Roles with Multiple Resources

In the previous examples, we always assumed that there was one resource in each role. The most obvious resource extension is to model that there are multiple resources in a role. Figure 1.22 shows a simple example of this. In this figure there is not one cassier, but there are five. For the example, we also increased the arrival rate of customers (otherwise it does not make much sense to have five cassiers). In Figure 1.22 one customer arrives every two minutes instead of every 10 minutes.

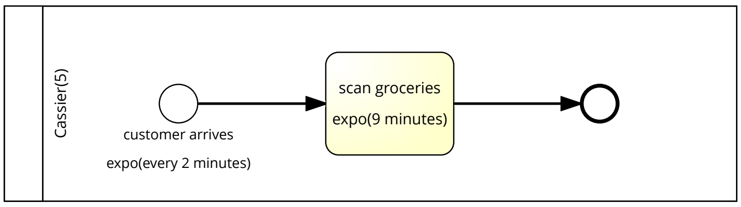

Figure 1.22: a role with multiple resources.

Increasing the number of resources per role is not only easy to model in BPMN, but also in SimPN. We simply add more cassier values to the cassier variable, one for each cassier that we need. In the running example, we can change the number of cassiers as follows:

```python
cassier = shop.add_var("cassier")
for i in range(1, 5):
  cassier.put("r" + str(i))
```

1.10.2 Multiple Queues

If we simply increase the capacity of a role, the resources in this role share a queue. Customers will arrive in the queue and each resource - when it becomes available - will serve the customer at the head of the queue. Conversely, resources can also have their own queue. In that case, when a customer arrives, it needs to make a choice of the queue in which it will take place and consequently the resource by which it will be served.

While this can be modeled in BPMN, it is difficult to do so and leads to replication of tasks. Therefore, we will not model the construct in BPMN, but instead look immediately at the SimPy code for modeling this construct.

```python
to_choose = shop.add_var("to choose")

waiting_1 = shop.add_var("cassier 1 queue")
waiting_2 = shop.add_var("cassier 2 queue")
cassier_1 = shop.add_var("cassier 1")
cassier_2 = shop.add_var("cassier 2")
cassier_1.put("c1")
cassier_2.put("c2")

to_done = shop.add_var("to done")

```

```
12 BPMNStartEvent(shop, [], [to_choose], "arrive", lambda: exp(1/6))
13
14 def choose(c, c1_queue, c2_queue):
15   if len(c1_queue) < len(c2_queue):
16     c1_queue.append(SimToken(c))
17   else:
18     c2_queue.append(SimToken(c))
19   return [c1_queue, c2_queue]
20 shop.add_event([to_choose, waiting_1.queue, waiting_2.queue],
21                [waiting_1.queue, waiting_2.queue], choose)
22
23 BPMNTask(shop, [waiting_1, cassier_1], [to_done, cassier_1],
24          "scan_groceries_1",
25          lambda c, r: [SimToken((c, r), delay=exp(1/9))])
26 BPMNTask(shop, [waiting_2, cassier_2], [to_done, cassier_2],
27          "scan_groceries_2",
28          lambda c, r: [SimToken((c, r), delay=exp(1/9))])
29
30 BPMNEndEvent(shop, [to_done], [], "leave")
```

To model this construct, first we need to model multiple cassiers and cassier queues. A sensible strategy for the customer is to look at the number of customers in front of it in the queues and then select the queue with the lowest number. The code illustrates this strategy.

We model the problem as a choice that the customer makes. First, we model two tasks, one for each queue, each with its own cassier and queue (line 3-8). Now we have to choose for an arriving customer, which of the two queues the customer goes into. We model this as a choice (line 14-21). When a customer arrives it is put into the `to_choose` variable. The `choose` event takes a customer from that variable as input and it also takes both queues, `waiting_1.queue` and `waiting_1.queue` as input. Note that we are handling the variables as queues now, such that the `choose` event takes entire queues rather than a single customer from the queue (also see Section 1.5). Now that we have the complete queue in the `choose` behavior (line 14-19), we can compare the queue lengths and add the customer to the appropriate queue (line 15-18). Finally, we have to put the queues back, with the customer attached (line 19).

1.10.3 Roles with Priority Queues

By default, values are taken from variables in a first-come-first-serve (FCFS) manner, which also means that customers are handled by resources in a first-come-first-serve manner. However, in practice there are many other possible strategies according to which customers are handled when then arrive in a waiting queue. In SimPN these strategies can be implemented by changing the order in which values are taken from variables, using the `priority` parameter of a variable, which must be a function that takes a token (consisting of a timestamp and a value) and returns a number for that token. Tokens with a lower number are then used before tokens with a higher number. This is illustrated in the code below.

```
1  scan_queue = shop.add_var("scan queue",
2                            priority=lambda token: token.value[1])
3
4  ...
5
6  BPMNStartEvent(shop, [], [scan_queue], "customer_arrived",
7                 lambda: exp(1/10),
8                 behavior=lambda: [SimToken(randint(1, 2))])
9
10 BPMNTask(shop, [scan_queue, cassier], [done, cassier],
11          "scan_groceries",
12          lambda c, r: [SimToken((c, r), delay=exp(1/9))])
13
14 BPMNEndEvent(shop, [done], [], "done")
```

In this example, arriving customers are given an additional value that is either 1 or 2 as data (line 6-8). This value represents the priority of the customers. Customers with a priority of 1 should be handled before customers with a priority of 2. Remember that customers always have a their unique identifier as first element. This random number therefore becomes their second element. Now we change the priority of the `scan queue` variable, by changing the priority function (line 1-2). The priority function takes a token and returns the second element of that token's value as its priority. Consequently, customers with the number 1 will now be taken before customers with the number 2. Note that the default priority function equals `lambda token: token.time`, meaning that tokens that were placed in the queue with a lower timestamp are handled first.

The code above generates the priority as a data-element of a customer when it arrives. It is of course quite possible to combine the priority construct with the different case types construct from Section 1.9.6. By doing so customers with different levels of priority, such as preferred customers and regular customers, can be generated separately according to their own arrival rate.

The priority construct can be used to model other concepts than 'priority customers' as well. We can, for example, easily model the earliest-deadline-first (EDF) strategy, in which customers have a deadline and customers that are closer to their deadline should always be handled before customers for which the deadline is further into the future. Assuming each customer has a deadline that indicates some moment in simulation time, we can model the EDF strategy by using that as the priority with which values are taken from the queue variable. Remember that in lower numbers have priority over higher numbers, so this would lead to customers with a deadline that is lower (i.e. closer in time) to be handled before customers with a deadline that is higher (i.e. further into the future).

1.10.4 Resources with Limited Availability

Resources may not always be available to handle customers: they may work on a schedule, take breaks at regular intervals, or perform activities in other processes as well.

We can model this by also modeling the behavior of resources. Until now, we took resources from a variable in which they were always available (if they were not busy processing a task). We can also take resources from that variable to send them on a break or make them do something else. The code below illustrates this.

```python
time = shop.var("time")
cassier = shop.add_var("cassier")
cassier.put(("r1", 0))

...

BPMNTask(shop, [scan_queue, cassier], [done, cassier_break_check],
         "scan_groceries",
         lambda c, r: [SimToken((c, r), delay=1)])

def check_break(c, time):
  if time > c[1] + 2:
    return [SimToken(c, delay=2)]
  else:
    return [SimToken(c)]
shop.add_event([cassier_break_check, time], [cassier], check_break)
```

Note that this code makes use of the reserved variable `time` that always contains the current simulation time. Furthermore, it associates the cassier with an identifier (`r1`) as well as the time at which it last returned from a break (initially this time equals 0). The code then defines a task that takes cassiers from the `cassier` variable as usual, but it puts cassiers back into the `cassier_break_check` variable. The rest of the code presents what happens with cassiers from then on. If the current time is greater than the last time a cassier went on a break plus 2 (i.e., the cassier went on a break longer than 2 time units ago), the cassier is put back into the `cassier` variable, but with a delay of 2 time units, effectively making it unavailable for processing tasks (i.e., on a break) for 2 time units. Otherwise, the cassier is simply put back into the `cassier` variable, making it available again immediately.

1.10.5 Resources with Limited Queue Space

Resources can have limited queue space. If that queue space is exceeded, customers may leave or other mechanisms may be put into place to handle customers efficiently during peak times. The code below shows an example in which customers simply leave when they arrive and they see that there are already 5 customers waiting.

```python
BPMNStartEvent(shop, [], [to_check_queue],
               "start", lambda: exp(1/9))

def check_queue_length(customer, complete_queue):
    if len(complete_queue) < 5:
        complete_queue.append(SimToken(customer))
        return [complete_queue, None]
    else:
        return [complete_queue, SimToken(customer)]
shop.add_event([to_check_queue, waiting.queue],
11                [waiting.queue, to_leave], check_queue_length)
12
13 BPMNTask(shop, [waiting, cassier], [done, cassier],
14          "scan_groceries",
15          lambda c, r: [SimToken((c, r), delay=exp(1/9))])
16
17 BPMNEndEvent(shop, [done], [], "complete")
18 BPMNEndEvent(shop, [to_leave], [], "leave")
```

The code uses a choice construct and operates on the queue variable as a whole rather than a single value from the variable (see Section 1.5). The choice that is made by the customer in the `to_check_queue` variable is whether to go into the queue, in case there are less than 5 people waiting, or to leave otherwise. To that end, the `check_queue_length` event takes a customer from the `to_check_queue` variable and the entire queue of waiting customers `waiting.queue`. It then checks the length of the queue and either places the customer in it or puts the customer in the `to_leave` variable (line 3-10). In any case the queue must be placed back into the `waiting` variable with or without the customer.