Aivika:
Computation-based Modeling and Simulation in Haskell
(Draft)

David E. Sorokin <david.sorokin@gmail.com>,
Yoshkar-Ola, Mari El, Russia

October 29, 2017

Contents

Introduction

To be written...

Introduce the DES abbreviation for discrete event simulation.

Part I

Sequential Simulation

In the first part the sequential simulation is considered. It covers the most of use cases. The corresponding implementation in Aivika is fastest and simplest. Here the simulation run is single-threaded, but the operations occur one by one in sequential order. Moreover, the model is single and it has no branches.

Other parts consider the parallel distributed simulation and nested simulation.

Chapter 1

Getting Started

The installation instructions are described in appendix A of this document. You can use Aivika on all main three platforms: Windows, OS X and Linux. All examples provided in this document must work everywhere, using possibly different charting back-ends. It is assumed below that your project depends on packages `aivika`, `aivika-experiment` and `aivika-experiment-chart`.

But before we start creating simulation models, we have to introduce some high-level concepts.

1.1 Simulation

In Aivika we can treat the simulation as a polymorphic function of the simulation run:

```
newtype Simulation a = Simulation (Run -> IO a)
```

Using the variable type allows us to create different entities within the simulation. As you can see, `Simulation` is a monad.

Given the specified simulation `Specs`, we can launch a simulation, where Aivika will create a simulation `Run` and then launch already the computation to receive the result:

```
runSimulation :: Simulation a -> Specs -> IO a
```

The simulation specs can contain the information about the start time and final time of modeling. Since Aivika also allows us to integrate the differential equations, we must provide the integration time step and method regardless of whether they are actually used. Also the specs can define the random number generator that we can use in the model.

```
data Specs = Specs { spcStartTime :: Double,
                     -- ^ the start time
                     spcStopTime :: Double,
                     -- ^ the stop time
                     spcDT :: Double,
                     -- ^ the integration time step
                     spcMethod :: Method,
                     -- ^ the integration method
```

```
    spcGeneratorType :: GeneratorType
    -- ^ the random number generator type
}
```

For the sake of simplicity, we will specify the 4th-order Runge-Kutta method and a default random number generator, which became quite fast in the recent versions to note.

For example, we can define the following simulation specs:

```
specs = Specs { spcStartTime = 0,
                spcStopTime = 13,
                spcDT = 0.01,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }
```

The mentioned above `Run` type is quite implementation dependent and it is hidden in depths of API from direct using by the modeler. At least, it must contain the specs provided so that they could be passed in to every part of the `Simulation` computation.

To allow running the Monte-Carlo simulation, the `Run` value must also contain the information about how many simulation runs are launched in parallel as well as it must contain the current simulation run index. Then we can run the specified number of parallel simulations, where each simulation run will be distinguished by its index as well as it will contain its own instances of the event queue and random number generator:

```
runSimulations :: Simulation a -> Specs -> Int -> [IO a]
```

The main idea is that many simulation models can be ultimately reduced to the `Simulation` computation. Hence they can be trivially simulated using the mentioned above run functions by the specified specs.

1.2 External Parameters

In practice many models depend on external parameters, which is useful for providing the Sensitivity Analysis.

To represent such parameters, Aivika uses almost the same definition that is used for representing the `Simulation` computation.

```
newtype Parameter a = Parameter (Run -> IO a)
```

A key difference between these two computations is that the parameter can be memoized before running the simulation so that the resulting `Parameter` computation would return a constant value within every simulation run and then its value would be updated for other runs (in a thread-safe way).

```
memoParameter :: Parameter a -> IO (Parameter a)
```

We usually have to memoize the parameter if its computation is not pure and it depends on the `IO` actions such as reading an external file or generating a random number.

It is natural to represent the simulation specs as external parameters when modeling.

```
starttime :: Parameter Double
stoptime :: Parameter Double
dt :: Parameter Double
```

Since we provide the random number generator with the simulation specs, it is also natural to generate the random numbers within the `Parameter` computation.

```
randomUniform :: Double -> Double -> Parameter Double
randomNormal :: Double -> Double -> Parameter Double
randomExponential :: Double -> Parameter Double
randomErlang :: Double -> Int -> Parameter Double
randomPoisson :: Double -> Parameter Int
randomBinomial :: Double -> Int -> Parameter Int
```

There are other built-in random number distributions. Please refer to the corresponding documentation.

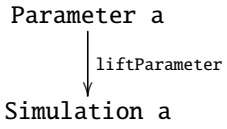
To support the Design of Experiments (DoE), Aivika has two additional computations that just return the corresponding fields from the simulation `Run` defining the current simulation run index and the total run count, respectively.

```
simulationIndex :: Parameter Int
simulationCount :: Parameter Int
```

An arbitrary parameter can be converted to the `Simulation` computation with help of the following function, which is actually defined in Aivika with help of a type class.

```
class ParameterLift m where
  liftParameter :: Parameter a -> m a

instance ParameterLift Simulation
```

It allows using parameters within the simulation:



1.3 Ordinary Differential Equations

In Aivika there is the `Point` type to represent a modeling time point within the current simulation run. Based on this type, we can define a polymorphic time varying function, which would be suitable for approximating the integrals.

The corresponding monadic computation is called `Dynamics` to emphasize the fact that it can model some dynamic process defined usually with help of ordinary differential equations (ODEs) and difference equations of System Dynamics.

```
newtype Dynamics a = Dynamics (Point -> IO a)
```

Since the modeling time is passed in to every part of the `Dynamics` computation, it is natural to have the following computation that returns the current time.

```
time :: Dynamics Double
```

There are different functions that allow running the `Dynamics` computation within the simulation: in the start time, in the final time, in all integration time points and in arbitrary time points defined by their numeric values.

```
runDynamicsInStartTime :: Dynamics a -> Simulation a
runDynamicsInStopTime :: Dynamics a -> Simulation a
runDynamicsInIntegTimes :: Dynamics a -> Simulation [IO a]

runDynamicsInTime :: Double -> Dynamics a -> Simulation a
runDynamicsInTimes :: [Double] -> Dynamics a -> Simulation [IO a]
```

1.3.1 Integrals

A key feature of the `Dynamics` computation is that it allows approximating the integral by the specified derivative and initial value:

```
integ :: Dynamics Double -> Dynamics Double -> Simulation (Dynamics Double)
```

The second parameter of the function might be a pure value, but using a computation here is more useful for practice as it allows direct referring to the initial value of the integral when defining the differential equations.

The point is that the ordinary differential and difference equations can be defined declaratively almost as in maths and as in many commercial simulation software tools of System Dynamics such as Vensim[17], ithink/Stella[6], Berkeley-Madonna[7] and Simtegra MapSys[1].

To create an integral, Aivika has to allocate an internal array to store the approximated values in the integration time points. It performs this side effect within the `Simulation` computation, where it has an access to the simulation specs.

Moreover, Aivika allows treating the parameterized `Dynamics` type as a numeric type, which greatly simplifies the definition of differential and difference equations as it will demonstrated below.

```
instance (Num a) => Num (Dynamics a)
```

For example, we can rewrite a model from the 5-Minute Tutorial of Berkeley-Madonna[7] with the following equations.

$$\begin{aligned}
\dot{a} &= -ka \times a, \qquad a(t_0) = 100, \\
\dot{b} &= ka \times a - kb \times b, \qquad b(t_0) = 0, \\
\dot{c} &= kb \times b, \qquad c(t_0) = 0, \\
ka &= 1, \\
kb &= 1.
\end{aligned}$$

Let us return the integral values in the final simulation time. In the same way, we could return the integral values in arbitrary time points, which we would specify by using other run functions.

[1] In the past the author developed visual simulation software tool `Simtegra MapSys`, but the software is unfortunately not available for the wide audience at time of preparing this document anymore.

```
{-# LANGUAGE RecursiveDo #-}

import Simulation.Aivika
import Simulation.Aivika.SystemDynamics

model :: Simulation [Double]
model =
  mdo a <- integ (- ka * a) 100
      b <- integ (ka * a - kb * b) 0
      c <- integ (kb * b) 0
      let ka = 1
          kb = 1
      runDynamicsInStopTime $ sequence [a, b, c]
```

Here we base on the fact that the `Simulation` type is `MonadFix` and hence it supports the recursive *do*-notation.

Now we can run the model using the 4th order Runge-Kutta method.

```
specs = Specs { spcStartTime = 0,
                spcStopTime = 13,
                spcDT = 0.01,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

main =
  runSimulation model specs >>= print
```

Had it been defined in the *Main.hs* file, we would receive the following results by running the simulation in the Terminal window[2]:

```
$ runghc Main.hs
[2.260329409450236e-4,2.938428231048658e-3,99.99683553882805]
```

The difference equations can be defined in a similar manner. The reader can find an example in the Aivika distribution.

Regarding the arrays and vectors of integrals, they are created naturally in Haskell. No special support is required. Only we need to use also the recursive *do*-notation to define an array if it has a loopback. The corresponding example is provided in the Aivika distribution too.

It is worth noting that we can embed external functions in the differential equations using the *do*-notation. It is possible thanks to the fact that the `Simulation` and `Dynamics` types are monads, because of which the numeric integration is rather slow, though.

1.3.2 Memoization and Side Effects

Moreover, there are helper functions that allow embedding external functions having a side effect. These helper functions order the calculations in the integration time points and use an interpolation in other time points.

For example, one of these functions is used in the mentioned above `integ` function for integrating.

[2]Here and below the `runghc` command is used in the examples, which implies that Aivika is installed with help of Cabal. But the same examples can be launched with help of Stack too, but only we had to create the corresponding Stack project. Stack seems to be a more preferable choice for the production code as it much less depends on the library changes.

```
memoDynamics :: Dynamics e -> Simulation (Dynamics e)
memo0Dynamics :: Dynamics e -> Simulation (Dynamics e)
```

The both functions memoize and order the resulting `Dynamics` computation in the integration time points. When requesting for a value in another time point, the both functions apply the simplest interpolation returning the value calculated in the closest less integration time point. But the functions behave differently, when integrating the equations with help of the Runge-Kutta method.

The `Point` type contains the additional information to distinguish intermediate integration time points used by the method. While the `memoDynamics` function memoizes the values in these intermediate time points, the second function `memo0Dynamics` just ignore these points applying the interpolation.

Therefore, the first memoization function is used by the `integ` function. In all other cases the second memoization function is more preferable as it is more efficient and consumes less memory.

Regarding the `Point` type, it is implementation-dependent. Like the `Run` type it is hidden from direct using by the modeler. The definition may change in the future without affecting the rest API.

1.3.3 Stochastic Equations

The considered above simulation monads are imperative as they are based on the `IO` monad. Namely this fact allows using the random number generator within the simulation. Therefore, the ordinary differential equations can be stochastic. The Aivika library provides useful helper random functions similar to those ones that are used in other software tools of System Dynamics.

```
memoRandomUniformDynamics ::
  Dynamics Double -> Dynamics Double -> Simulation (Dynamics Double)

memoRandomNormalDynamics ::
  Dynamics Double -> Dynamics Double -> Simulation (Dynamics Double)

memoRandomExponentialDynamics ::
  Dynamics Double -> Simulation (Dynamics Double)

memoRandomErlangDynamics ::
  Dynamics Double -> Dynamics Int -> Simulation (Dynamics Double)

memoRandomPoissonDynamics ::
  Dynamics Double -> Simulation (Dynamics Int)

memoRandomBinomialDynamics ::
  Dynamics Double -> Dynamics Int -> Simulation (Dynamics Int)
```

They are based on the mentioned earlier random functions returning the `Parameter` computation, but only these functions memoize the generated values in the integration time points and apply the interpolation in all other time points. These functions are designed to be used in the differential and difference equations.

1.3.4 Difference Equations

Regarding the difference equations, they can be built like differential ones. The following function returns an accumulated sum represented as the `Dynamics` computation by the specified difference and initial value.

```
diffsum :: (Num a, Unboxed a)
           => Dynamics a
           -> Dynamics a
           -> Simulation (Dynamics a)
```

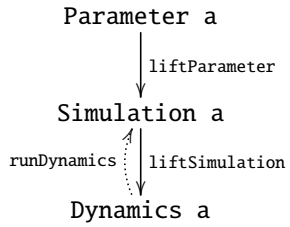
Here the `Unboxed` type class specifies what types can represent *unboxed* values for more efficient storing in memory. Its definition can be found in the Aivika library.

1.3.5 Lifting Computations

Finally, we can convert an arbitrary `Simulation` computation to the `Dynamics` one.

```
class SimulationLift m where
  liftSimulation :: Simulation a -> m a

instance SimulationLift Dynamics
instance ParameterLift Dynamics
```

It literally means that we can use external parameters and computations defined on level of the simulation run in the ordinary differential and difference equations:



1.4 Simulation Experiment

Besides the modeling constructs, there are other things which the simulation library should provide. To validate the model or to analyze it, Aivika allows us to automate the process of displaying the most important simulation results.

The simulation library can save the results in CSV files that can be then opened in the Office application or R statistics tool for further analysis. Also the library can plot charts and histograms by collected statistics.

One of the important charts is so called the *deviation chart* that shows the trend and confidence interval by rule *3-sigma*. There are also *time series* and *XY chart*, which Aivika plot for each run, while the deviation chart is cumulative and it is displayed for the entire Monte-Carlo simulation experiment, which may consist of thousands of simulation runs.

When running the simulation experiment, Aivika creates a Web page containing file *index.html* and the corresponding auxiliary files. Then you can open the Web page in your favorite Internet browser to observe the simulation results.

This approach actually allows running thousands of simulation runs within one experiment, when only necessary data are kept in memory. At the same time, the Internet browser becomes a tool for displaying the results.

1.4.1 Returning Results from Model

At first, we have to prepare the results of simulation. We associate each variable with some `String` name using the `resultSource` function. Then we collect such associations and return them as a value of type `Results` within the `Simulation` computation.

Our system of ordinary differential equations from the previous section can be rewritten in the following way.

```
{-# LANGUAGE RecursiveDo #-}

import Simulation.Aivika
import Simulation.Aivika.SystemDynamics
import Simulation.Aivika.Experiment

model :: Simulation Results
model =
  mdo a <- integ (- ka * a) 100
      b <- integ (ka * a - kb * b) 0
      c <- integ (kb * b) 0
      let ka = 1
          kb = 1
      return $ results
        [resultSource "t" "time" time,
         resultSource "a" "variable A" a,
         resultSource "b" "variable B" b,
         resultSource "c" "variable C" c]
```

Now we can experiment with this model. For example, we can launch a single simulation run to see the results of simulation in final time.

```
specs = Specs { spcStartTime = 0,
                spcStopTime = 13,
                spcDT = 0.01,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

main =
  printSimulationResultsInStopTime
  printResultSourceInEnglish
  model specs
```

It will print the following information:

```
----------

-- simulation time
t = 13.0

-- time
```

```
t = 13.0

-- variable A
a = 2.260329409450236e-4

-- variable B
b = 2.938428231048658e-3

-- variable C
c = 99.99683553882805
```

Here the time is printed twice. The library function always prints the simulation time. Also we bind name `t` with the current time. Note that the comments are different.

In the same manner we could print the results in the integration time points. Also we can show the information either in Russian or English.

The point is that we can return the variables of different nature in the `Results` value. As we will see, we can return resources, queues, servers, arrays, lists etc. We can customize it and return our own data type values in case of need.

An intriguing thing is that the `Results` value can actually be a source of simulation results for different types of analysis, which can be more than just printing in the terminal. So, we can plot charts and histograms, saves the results in files. Let us see how we can do it.

1.4.2 Experiment Definition

Now we define an `Experiment` object specifying the simulation specs and a number of runs. By specifying the number of runs greater than one, we actually receive a Monte-Carlo simulation.

```
import Simulation.Aivika.Experiment

experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1,
    experimentTitle = "Chemical Reaction",
    experimentDescription = "Chemical Reaction as described in " ++
                            "the 5-minute tutorial of Berkeley-Madonna" }
```

Then we define the series and generators that already know how to render the results. Note that we refer to the variables by their `String` names that we used when specifying the result sources by calling function `resultSource` in the model.

```
t = resultByName "t"
a = resultByName "a"
b = resultByName "b"
c = resultByName "c"

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView $ defaultTableView {
     tableSeries = t <> a <> b <> c },
```

```
  outputView $ defaultTimeSeriesView {
    timeSeriesTitle = "Time Series",
    timeSeriesLeftYSeries = a <> b <> c } ]
```

Here we specify that we want to save the results in the CSV file (table) and plot the time series chart.

Then we render a Web page by the specified model and experiment using our generators. This page will contain the chart and a link to the corresponding CSV file.

1.4.3 Charting

There is a choice. We plot the chart and hence we have to select the charting back-end. There are two back-ends: Cairo-based and Diagrams-based. The choice can depend on your platform that can support only one of the back-ends, or the both. Please read Appendix B of this document for more detail.

Cairo-based Charting Back-end

Using the Cairo back-end, we import the necessary libraries and start the simulation experiment in the following way:

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Cairo

import Graphics.Rendering.Chart.Backend.Cairo

import Model
import Experiment

main =
  do let r0 = CairoRenderer PNG
         r  = (WebPageRenderer r0 experimentFilePath)
     runExperimentParallel experiment generators r model
```

Diagrams-based Charting Back-end

Choosing the Diagrams back-end, we import other libraries and the start code looks slightly different.

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Diagrams

import Graphics.Rendering.Chart.Backend.Diagrams

import Model
import Experiment

main =
  do fonts <- loadCommonFonts
     let r0 = DiagramsRenderer SVG (return fonts)
         r  = WebPageRenderer r0 experimentFilePath
     runExperimentParallel experiment generators r model
```

Given the model, experiment and generators, we run the simulation experiment using one of the charting back-ends.

1.4.4 Running Simulation Experiment

When running the experiment with help of the Cairo back-end, we receive the following output in the macOS terminal[3]:

```
$ ghc -O2 -threaded MainUsingCairo.hs
$ ./MainUsingCairo +RTS -N
Updating directory experiment
Generated file experiment/Table(1).csv
Generated file experiment/TimeSeries(1).png
Generated file experiment/index.html
```

It means that the application created a new directory `experiment` containing the Web page, which we can open in the Internet browser.

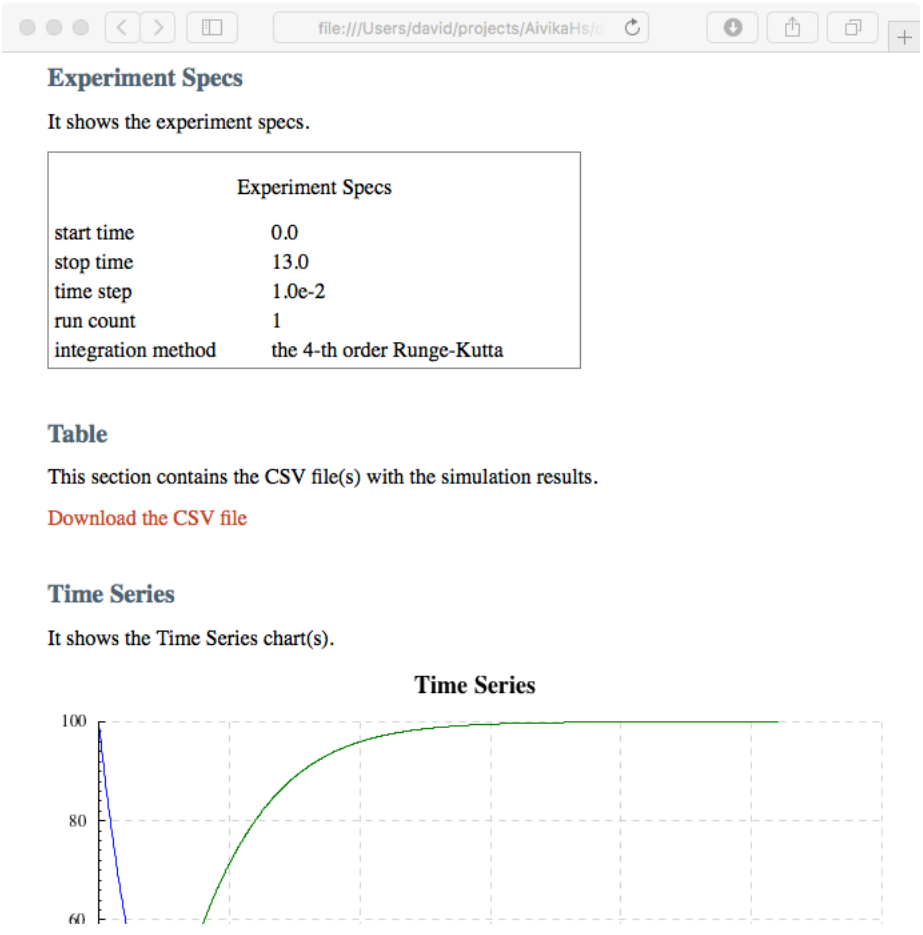


Figure 1.1: The rendered simulation experiment in the Internet browser.

As demonstrated in figure 1.1, the Web page shows the simulation experiment specs, the time series chart and provides with a hyper-link to the CSV

[3]We could run the simulation by typing in the terminal "`runghc MainUsingCairo.hs`", but then the code would be interpreted. We want to compile it with optimization to achieve the highest speed available.

file with the results. The chart is shown on figure 1.2 in more detail. We will receive similar results on Windows and Linux too, whatever charting back-end we will use.

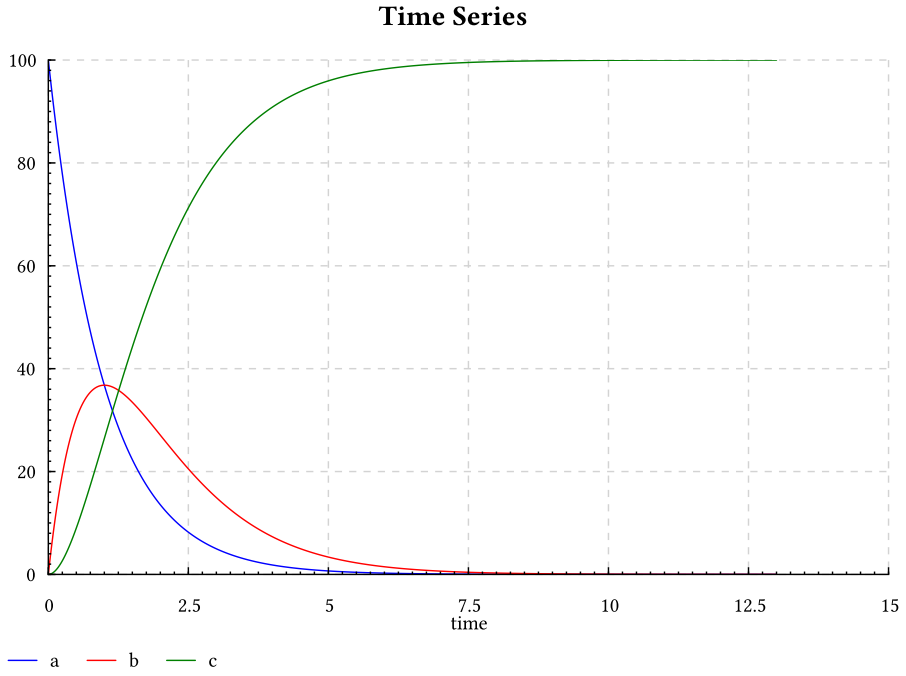


Figure 1.2: The time series chart for chemical reaction.

Chapter 2

Discrete Event Simulation

Earlier we saw how Aivika can be useful for integrating the ordinary differential and difference equations, but Aivika is mainly focused on discrete event simulation. The library supports many paradigms of DES, but their implementation is based on using the event queue. In other words, speaking of Aivika, we can say that other paradigms of DES are reduced ultimately to the event oriented paradigm. It is true for the process-oriented and activity-oriented paradigms, at least in that sense how they are represented in Aivika.

2.1 Event-oriented Simulation

Under the *event-oriented* paradigm[11, 8] of DES, we put all pending events in the priority queue, where the first event has the minimal activation time. Then we sequentially activate the events removing them from the queue. During such an activation we can add new events. This scheme is also called *event-driven*.

Aivika uses almost the same time-varying function for the event-oriented simulation, which we used for approximating the integrals with help of the `Dynamics` monad.

```
newtype Event a = Event (Point -> IO a)
```

The difference is that Aivika strongly guarantees[1] on level of the type system of Haskell that the `Event` computation is always synchronized with the event queue. Here we imply that every simulation run has an internal event queue, which actually belongs to the `Run` type value we saw before.

A key feature of the `Event` monad is an ability to specify the event handler that should be actuated at the desired modeling time, when the corresponding event occurs.

```
enqueueEvent :: Double -> Event () -> Event ()
```

To pass in a message or some other data to the event handler, we just use a closure when specifying the event handler in the second argument.

[1] Actually, there is a small room in Aivika for some hacking that may break this strong guarantee.

The event cancellation can be implemented trivially. We create a wrapper for the source event handler and pass in namely this wrapper to the `enqueueEvent` function. Then the wrapper already decides whether it should call the underlying source event handler. We have to provide some means for notifying the wrapper that the source event handler must be cancelled. The Aivika library has the corresponded support.

The same technique of canceling the event can be adapted to implementing the timer and time-out handlers used in the agent-based modeling as it is described later.

To be involved in the simulation, the `Event` computation must be run explicitly or implicitly within the `Dynamics` computation. The most simple run function is stated below. It actuates all pending event handlers from the event queue relative to the current modeling time and then runs the specified computation.

```
runEvent :: Event a -> Dynamics a
```

There is a subtle thing related to the `Dynamics` computation. In general, the modeling time flows unpredictably within `Dynamics`, while there is a guarantee that the time is synchronized with the event queue and flows monotonically within the `Event` computation.

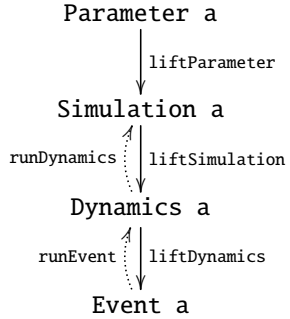
Other two run functions are destined for the most important use cases, when we can run the input computation directly within `Simulation` in the initial and final modeling time points, respectively. These two functions apply the `runEvent` function.

```
runEventInStartTime :: Event a -> Simulation a
runEventInStopTime :: Event a -> Simulation a
```

Following the rule, an arbitrary `Dynamics` computation can be transformed to the `Event` computation. As before, the corresponding function is defined in a type class.

```
class DynamicsLift m where
  liftDynamics :: Dynamics a -> m a

instance DynamicsLift Event
instance SimulationLift Event
instance ParameterLift Event
```

It means that integrals, external parameters and computations on level of the simulation run can be directly used in the event-oriented simulation.



2.2 Mutable Reference

Many DES models need a mutable reference. Since Haskell is a pure functional programming language, all side effects must be expressed explicitly in the type signatures. Mutable references require such effects.

In the Haskell standard libraries, `IORef` is the standard mutable reference. Aivika introduces a very similar but strict `Ref` reference, where all computations are synchronized with the event queue.

```
data Ref a

newRef :: a -> Simulation (Ref a)

readRef :: Ref a -> Event a
writeRef :: Ref a -> a -> Event ()
modifyRef :: Ref a -> (a -> a) -> Event ()
```

The simulation model should use the `Ref` type instead of `IORef` within the `Simulation` computation whenever possible.

Especially, it is important for the distributed simulation, which is also supported by Aivika, because using any `IO` action would mean a synchronization of the global virtual time among all logical processes involved in the distributed simulation. But the `Ref` type is much and much faster as it supports roll-backs and has no need in heavy synchronization.

Regarding the nested simulation supported by Aivika too, the `Ref` type has an additional feature. Such a reference can be efficiently changed in the derivative nested branches of simulation without affecting the reference value in the descending branches of the same simulation model. But let us return to describing the basics of Aivika.

2.3 Example: Event-oriented Simulation

The Aivika distribution contains examples of using the mutable references in the DES models, one of which is provided below. The task itself is described in the documentation of SimPy[8].

> There are two machines, which sometimes break down. Up time is exponentially distributed with mean 1.0, and repair time is exponentially distributed with mean 0.5. There are two repairpersons, so the two machines can be repaired simultaneously if they are down at the same time. Output is long-run proportion of up time. Should get value of about 0.66.

The corresponding model is as follows. It is worth nothing that the represented definition is not simplest. As you will see further in section 2.6, there are other ways. But we need to introduce some new concepts before we could rewrite the model.

```
import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
```

```
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 1000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation Results
model =
  do totalUpTime <- newRef 0.0

     let machineBroken :: Double -> Event ()
         machineBroken startUpTime =

           do finishUpTime <- liftDynamics time
              modifyRef totalUpTime (+ (finishUpTime - startUpTime))
              repairTime <-
                liftParameter $
                randomExponential meanRepairTime

              -- enqueue a new event
              let t = finishUpTime + repairTime
              enqueueEvent t machineRepaired

         machineRepaired :: Event ()
         machineRepaired =

           do startUpTime <- liftDynamics time
              upTime <-
                liftParameter $
                randomExponential meanUpTime

              -- enqueue a new event
              let t = startUpTime + upTime
              enqueueEvent t $ machineBroken startUpTime

     runEventInStartTime $
       do -- start the first machine
          machineRepaired
          -- start the second machine
          machineRepaired

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

     return $
       results
       [resultSource
        "upTimeProp"
        "The long-run proportion of up time (~ 0.66)"
        upTimeProp]

main =
  printSimulationResultsInStopTime
  printResultSourceInEnglish
  model specs
```

We model two machines, each of them has two states. Every state is represented as the `Event` computation. These computations can be registered as

event handlers. Actuating the handler, we change the state of the machine.

After running the example, we receive a desired result with some accuracy.

```
$ runghc MachRep1EventDriven.hs
----------

-- simulation time
t = 1000.0

-- The long-run proportion of up time (~ 0.66)
upTimeProp = 0.6635359777536585
```

Frankly speaking, the use of the event-oriented paradigm may seem to be quite tedious. Aivika supports more high-level paradigms. Later it will be shown how the same task can be solved in a more elegant and simple way.

2.4 Variable with Memory

Sometimes we need to mix the ordinary differential equations with discrete event simulation. In most cases, it would be more simple and efficient to encode Euler's method directly within the `Event` computation. But if you still need to use complex differential equations defined within the `Dynamics` computation, then there is an approach that allows creating a combined simulation model. It is not free and has a cost, though.

For that, there is an analog of the mutable reference that would save the history of its values. Aivika defines the corresponding `Var` type. It has almost the same functions with similar type signatures that the `Ref` reference has.

```
data Var a

newVar :: a -> Simulation (Var a)

readVar :: Var a -> Event a
writeVar :: Var a -> a -> Event ()
modifyVar :: Var a -> (a -> a) -> Event ()
```

However, we can also use the variable in the differential and difference equations requesting for the *first* actual value for *each* time point with help of the following function, actuating the pending events if required.

```
varMemo :: Var a -> Dynamics a
```

The magic is as follows. The `Var` variable stores the history of changes. When updating the mutable variable, or requesting it for a value at new time point, the `Var` data object stores internally the value, which was first for the requested time point. Then it becomes constant within the simulation. Therefore, the computation returned by the `varMemo` function can be used in the differential and difference equations of System Dynamics.

On the contrary, the `readVar` function returns a computation of the *recent* actual value for the *current* simulation time point. This value is already destined to be used in the discrete event simulation as it is synchronized with the event queue. Such is the `Event` computation that must be synchronized with the event queue.

In case of need we can freeze temporarily the variable and receive its internal state: triples of time, the first and last values for the corresponding time value.

```
freezeVar :: Var a -> Event (Array Int Double, Array Int a, Array Int a)
```

The time values returned by this function are distinct and sorted in ascending order.

It is worth noting that the `Var` variable is slow. It would be a logical mistake to use `Var` for collecting statistics only. It would be rather inefficient. Moreover, it would consume a lot of memory. In Aivika there are special data structures that should be used for collecting statistics, though. See sections 4.1 and 4.2 for detail.

2.5 Process-oriented Simulation

Under the *process-oriented* paradigm[11, 8], we model simulation activities with help of a special kind of processes. We can explicitly suspend and resume such processes. Also we can request for, and release of, the resources implicitly suspending and resuming the processes in case of need.

Aivika actually supports the process-oriented simulation on different levels. So, there are streams of data and processors that operate on these streams, but they are considered further as well as GPSS-like blocks. Below is described a lower level, which is a foundation for the higher levels, nevertheless.

2.5.1 Discontinuous Processes

The discontinuous process is represented by the continuation-based monad `Process`. Moreover, there is an associated process identifier `ProcessId` type.

```
data Process a
data ProcessId

newProcessId :: Simulation ProcessId
```

We can run the process within the simulation with help of one of the next functions.

```
runProcess :: Process () -> Event ()
runProcessUsingId :: ProcessId -> Process () -> Event ()

runProcessInStartTime :: Process () -> Simulation ()
runProcessInStartTimeUsingId :: ProcessId -> Process () -> Simulation ()

runProcessInStopTime :: Process () -> Simulation ()
runProcessInStopTimeUsingId :: ProcessId -> Process () -> Simulation ()
```

If the process identifier is not specified then a new generated identifier is assigned at time of starting the process. Every process has always its own unique identifier.

```
processId :: Process ProcessId
```

A characteristic feature of the `Process` monad is that the process can be hold for the specified time interval through the event queue.

```
holdProcess :: Double -> Process ()
```

It allows modeling some activity by delaying the computation. After the time interval passes, the computation will resume its execution starting from another time point, the current time plus the specified time interval.

Nevertheless, the held process can be immediately interrupted and we can request for whether it indeed was interrupted. The information about this is stored until the next call of the `holdProcess` function.

```
interruptProcess :: ProcessId -> Event ()
processInterrupted :: ProcessId -> Event Bool
```

The interrupted in such a way process continues its execution as if the `holdProcess` function finished immediately. But there is another kind of interruption, which is called a *preemption*. The preempted process suspends temporarily and then resumes executing the `holdProcess` function from the point where it was preempted. Please see section 3.8 for detail.

It is worth noting to say more about the types of computations returned by the functions. The `Event` type of the result means that the computation executes immediately and it cannot be interrupted. On the contrary, the `Process` type of the result means that the corresponding computation may suspend, even forever. This is very important for understanding.

To passivate the process for indefinite time so that it could be reactivated later, we can use the following functions.

```
passivateProcess :: Process ()
processPassive :: ProcessId -> Event Bool
reactivateProcess :: ProcessId -> Event ()
```

Every process can be immediately cancelled, which is important for modeling some activities.

```
cancelProcessWithId :: ProcessId -> Event ()
cancelProcess :: Process a
processCancelled :: ProcessId -> Event Bool
```

Sometimes we need to run an arbitrary sub-process within the specified time-out, but please note that this is a very slow computation.

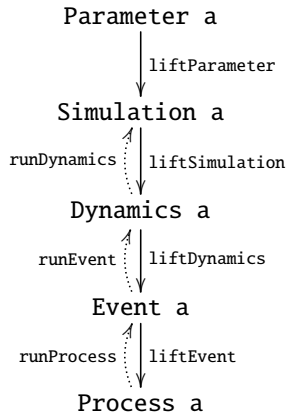
```
timeoutProcess :: Double -> Process a -> Process (Maybe a)
```

If the sub-process executes too long and exceeds the time limit, then it is immediately canceled and `Nothing` is returned within the `Process` computation. Otherwise; the computed result is returned right after it is received by the sub-process.

Every simulation computation we considered before can be transformed to the `Process` computation.

```
class EventLift m where
  liftEvent :: Event a -> m a

instance EventLift Process
instance DynamicsLift Process
instance SimulationLift Process
instance ParameterLift Process
```

It allows using integrals and external parameters as well as updating the mutable references and variables within the process-oriented simulation. It allows combining the event-oriented and process-oriented simulation parts of the model.



2.5.2 Spawning Processes in Parallel

Another process can be forked and spawn on-the-fly. If that process is not related to the current parent process in any way, then we can run the second process within the `Event` computation and then transform the result to the `Process` computation. There is no need to add a special function. It is enough to have `liftEvent` and one of the `Process` run functions.

```
liftEvent $ runProcess (p :: Process ())
```

But if the life cycle of the child process must be bound up with the life cycle of the parent process so that they would be canceled in some order if required, then we should use one of the next functions.

```
spawnProcess :: Process () -> Process ()
spawnProcess = spawnProcessWith CancelTogether

spawnProcessWith :: ContCancellation -> Process () -> Process ()
```

Here the first argument of the second function specifies how two processes are bound.

```
data ContCancellation =
    | CancelTogether
    | CancelChildAfterParent
    | CancelParentAfterChild
    | CancelInIsolation
```

The mentioned above `timeoutProcess` function uses `spawnProcessWith` to run the specified sub-process within time-out.

Also an arbitrary number of the `Process` computations can be launched in parallel and we can await the completion of all the started sub-processes to return the final result.

```
processParallel :: [Process a] -> Process [a]
```

2.5.3 Memoizing

The `Process` computation can be memoized so that the resulting process would always return the same value within the simulation run regardless of that how many times the resulting process was requested repeatedly. The source process is called once only.

```
memoProcess :: Process a -> Simulation (Process a)
```

Sometimes, it can be very useful to have this feature, for example, if we are going to clone the stream of data.

2.5.4 Exception Handling

The continuations, which the `Process` monad is based on, are known to be difficult for handling exceptions properly. Nevertheless, the author of Aivika adapted successfully the F# `Async` approach and added the corresponding functions to handle arbitrary exceptions within `IO`.

```
catchProcess :: Exception e => Process a -> (e -> Process a) -> Process a
finallyProcess :: Process a -> Process b -> Process a
throwProcess :: IOException -> Process a
```

There are similar functions for handling the exceptions within all other simulation monads considered in the document before.

It is worth noting that the `Process` computation is faster unless the exception handling is enabled. Right after you apply `catchProcess` or the `finallyProcess` function, additional checks are involved and hence the simulation becomes slower.

2.5.5 Random Process Delays

In models we often need to hold the process for a random time interval from the specified distribution. Aivika defines the corresponding helper functions. All them generate a time interval and pass in it to the `holdProcess` function like this.

```
randomUniformProcess min max =
  do t <- liftParameter $ randomUniform min max
     holdProcess t
     return t
```

There are two sibling functions for each distribution. The first function performs an action and returns the interval used. The second function performs the action only. Below are provided some of the functions. There are similar functions for other built-in distributions too.

```
randomUniformProcess :: Double -> Double -> Process Double
randomUniformProcess_ :: Double -> Double -> Process ()

randomNormalProcess :: Double -> Double -> Process Double
randomNormalProcess_ :: Double -> Double -> Process ()

randomExponentialProcess :: Double -> Process Double
randomExponentialProcess_ :: Double -> Process ()
```

```
randomErlangProcess :: Double -> Int -> Process Double
randomErlangProcess_ :: Double -> Int -> Process ()

randomPoissonProcess :: Double -> Process Int
randomPoissonProcess_ :: Double -> Process ()

randomBinomialProcess :: Double -> Int -> Process Int
randomBinomialProcess_ :: Double -> Int -> Process ()
```

Thus, the functions described in this section allow efficiently modeling quite complex activities. Nevertheless, the `Process` computation is still low-level. Aivika supports more high-level computations described further.

2.6 Example: Process-oriented Simulation

Let us return to the task that was solved in section 2.3 using the event-oriented paradigm. The problem statement is repeated here. It corresponds to the documentation of SimPy.

> There are two machines, which sometimes break down. Up time is exponentially distributed with mean 1.0, and repair time is exponentially distributed with mean 0.5. There are two repairpersons, so the two machines can be repaired simultaneously if they are down at the same time. Output is long-run proportion of up time. Should get value of about 0.66.

2.6.1 Returning Results from Model

Using the processes, we can solve the task in a more elegant way. At first, we have to write the model that would return the simulation results.

```
module Model(model) where

import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
meanRepairTime = 0.5

model :: Simulation Results
model =
  do totalUpTime <- newRef 0.0

     let machine :: Process ()
         machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)
              repairTime <-
                randomExponentialProcess meanRepairTime
              machine

     runProcessInStartTime machine
```

```
    runProcessInStartTime machine

    let upTimeProp =
          do x <- readRef totalUpTime
             y <- liftDynamics time
             return $ x / (2 * y)

    return $
      results
      [resultSource
       "upTimeProp"
       "The long-run proportion of up time (~ 0.66)"
       upTimeProp]
```

The reader can compare this model with the previous one. Conceptually, they do the same thing, use the same event queue and have the same behavior.

We could run the simulation by printing the results in terminal. But it would be more interesting to demonstrate how we can run the Monte-Carlo simulation to plot the deviation chart with confidence interval and to plot the histogram.

2.6.2 Experiment Definition

We define an experiment with 1000 simulation runs. The specs are the same as before.

```
module Experiment (experiment, generators) where

import Data.Monoid

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 1000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1000,
    experimentTitle = "MachRep1 Example",
    experimentDescription = "The simulation experiment." }

x = resultByName "upTimeProp"

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView defaultInfoView,
   outputView $ defaultDeviationChartView {
     deviationChartPlotTitle = "The Up-time Proportion Chart",
     deviationChartLeftYSeries = x },
   outputView $ defaultFinalHistogramView {
     finalHistogramPlotTitle = "The Up-time Proportion Histogram",
     finalHistogramSeries = x }]
```

Note how we refer to the variable by its name. The model deconstructs the simulation entities so that they could be returned in an unified way as a value of the `Results` type. Therefore, we have to restore the variables by their names.

Here we show the experiment specs, show the information about the variables, plot the deviation chart and then show the histogram by data collected in final time points for all simulation runs.

2.6.3 Charting

Applying the Cairo-based charting back-end, we run the experiment with help of the following code.

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Cairo

import Graphics.Rendering.Chart.Backend.Cairo

import Model
import Experiment

main =
  do let r0 = CairoRenderer PNG
         r  = (WebPageRenderer r0 experimentFilePath)
     runExperimentParallel experiment generators r model
```

Alternatively, we could run the experiment using the Diagrams-based charting back-end. Then we would receive the image files in the SVG format instead of PNG, but it should work on Windows.

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Diagrams

import Graphics.Rendering.Chart.Backend.Diagrams

import Model
import Experiment

main =
  do fonts <- loadCommonFonts
     let r0 = DiagramsRenderer SVG (return fonts)
         r  = WebPageRenderer r0 experimentFilePath
     runExperimentParallel experiment generators r model
```

You may notice that this code is the same as it was defined in section 1.4.3. Therefore, it won't be repeated anymore.

2.6.4 Running Simulation Experiment

The resulting deviation chart and histogram are shown on figures 2.1 and 2.2, respectively. On my not very new Macbook Pro, the Monte-Carlo simulation with 1000 (thousand) runs lasted for 2.847 seconds only (less than three seconds). It includes the plotting of charts.

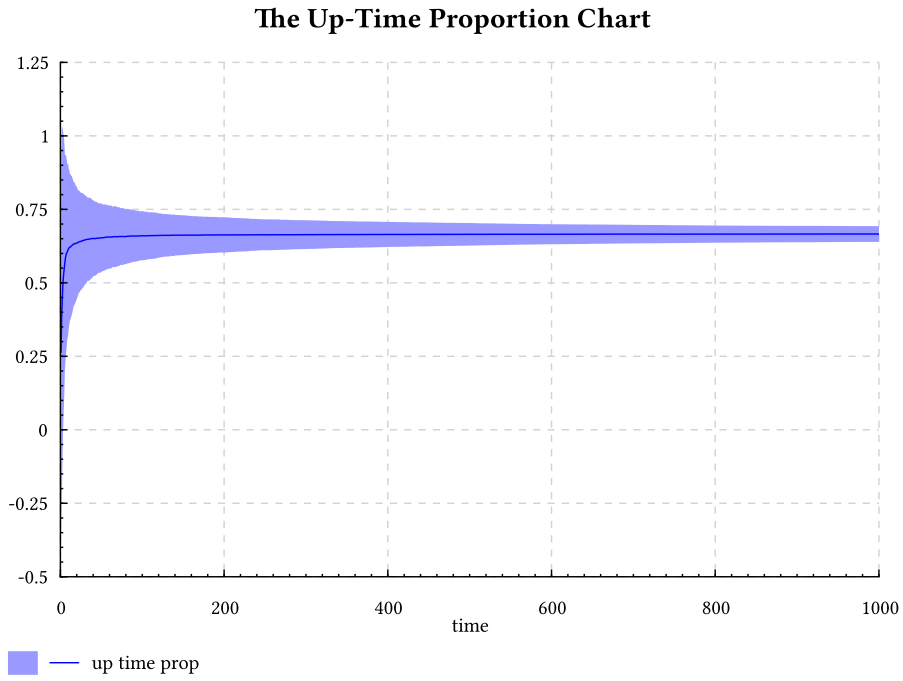


Figure 2.1: The deviation chart for the long-run proportion of up-time.

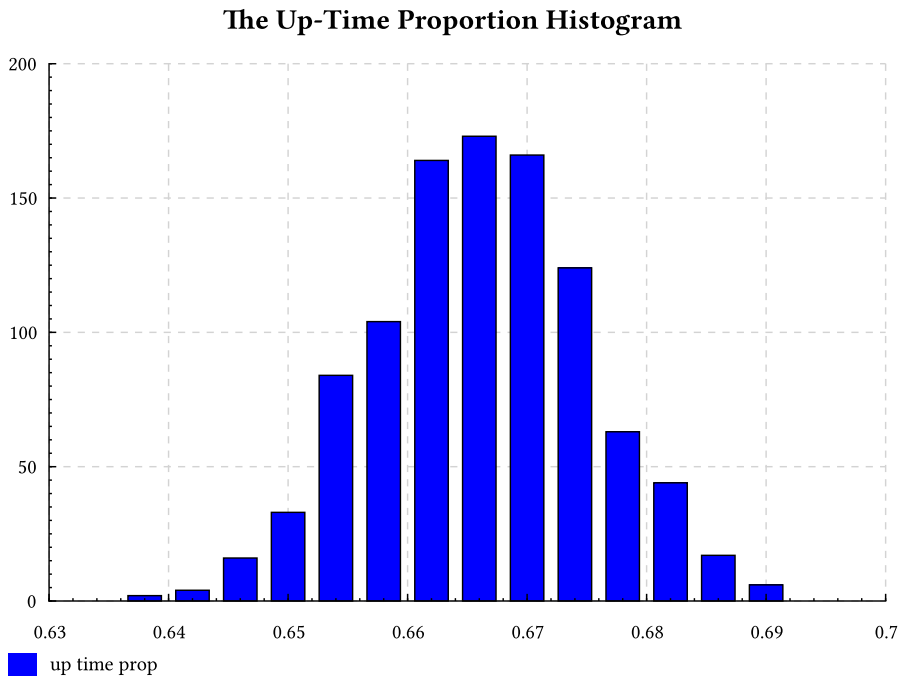


Figure 2.2: The histogram for the long-run proportion of up-time.

There is also another popular paradigm that can be applied to the discrete event simulation. It usually gives more rough simulation results as we have to scale the modeling time. The next two sections show how Aivika supports that paradigm and how we can apply it to solve the same task.

2.7 Activity-oriented Simulation

Under the *activity-oriented* paradigm[11, 8] of DES, we break time into tiny increments. At each time point, we look around at all the activities and check for the possible occurrence of events. Sometimes this scheme is called *time-driven*.

An idea is that we can naturally represent the activity as an `Event` computation, which we will call periodically through the event queue.

```
enqueueEventWithTimes :: [Double] -> Event () -> Event ()
enqueueEventWithTimes ts e = loop ts
  where loop []       = return ()
        loop (t : ts) = enqueueEvent t $ e >> loop ts
```

We can also use another predefined function that does almost the same, but only it calls the specified computation directly in the integration time points specified by the simulation specs.

```
enqueueEventWithIntegTimes :: Event () -> Event ()
```

Being defined in such a way, the activity-oriented simulation can be combined with the event-oriented and process-oriented ones. There is the corresponding example in the Aivika distribution, where the pit furnace[11, 16] is modeled.

2.8 Example: Activity-oriented Simulation

To illustrate the activity-oriented paradigm, let us take our old task that was solved in section 2.3 using the event-oriented paradigm and in section 2.6 using the process-oriented paradigm of DES. The problem statement is repeated here again. It corresponds to the documentation of SimPy.

> There are two machines, which sometimes break down. Up time is exponentially distributed with mean 1.0, and repair time is exponentially distributed with mean 0.5. There are two repairpersons, so the two machines can be repaired simultaneously if they are down at the same time. Output is long-run proportion of up time. Should get value of about 0.66.

Now the model looks quite cumbersome. Moreover, we have to scale the modeling time. The time points at which the events occur are not precise any more.

```
import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 1000.0,
                spcDT = 0.05,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation Results
model =
  do totalUpTime <- newRef 0.0

     let machine :: Simulation (Event ())
         machine =
           do startUpTime <- newRef 0.0

              -- a number of iterations when
              -- the machine works
              upNum <- newRef (-1)

              -- a number of iterations when
              -- the machine is broken
              repairNum <- newRef (-1)

              -- create a simulation model
              return $
                do upNum' <- readRef upNum
                   repairNum' <- readRef repairNum

                   let untilBroken =
                         modifyRef upNum $ \a -> a - 1

                       untilRepaired =
                         modifyRef repairNum $ \a -> a - 1

                       broken =
                         do writeRef upNum (-1)
                            -- the machine is broken
                            startUpTime' <- readRef startUpTime
                            finishUpTime' <- liftDynamics time
                            dt' <- liftParameter dt
                            modifyRef totalUpTime $
                              \a -> a +
                              (finishUpTime' - startUpTime')
                            repairTime' <-
                              liftParameter $
                              randomExponential meanRepairTime
                            writeRef repairNum $
                              round (repairTime' / dt')

                       repaired =
                         do writeRef repairNum (-1)
                            -- the machine is repaired
                            t'  <- liftDynamics time
                            dt' <- liftParameter dt
                            writeRef startUpTime t'
                            upTime' <-
```

```
                              liftParameter $
                              randomExponential meanUpTime
                            writeRef upNum $
                              round (upTime' / dt')

                        result | upNum' > 0      = untilBroken
                               | upNum' == 0     = broken
                               | repairNum' > 0  = untilRepaired
                               | repairNum' == 0 = repaired
                               | otherwise       = repaired
                  result

     -- create two machines with type Event ()
     m1 <- machine
     m2 <- machine

     -- start the time-driven simulation of the machines
     runEventInStartTime $
       -- in the integration time points
       enqueueEventWithIntegTimes $
       do m1
          m2

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

     return $
       results
       [resultSource
        "upTimeProp"
        "The long-run proportion of up time (~ 0.66)"
        upTimeProp]

main =
  printSimulationResultsInStopTime
  printResultSourceInEnglish
  model specs
```

The results are very similar to those ones we received earlier, although now the model is more rough and it is based on the different approach:

```
$ runghc MachRep1TimeDriven.hs
----------

-- simulation time
t = 1000.0

-- The long-run proportion of up time (~ 0.66)
upTimeProp = 0.6556000000000012
```

Nevertheless, the activity-oriented paradigm can be exceptionally useful for modeling some parts that are difficult to represent based on other simulation paradigms.

Chapter 3

Resources

Resources are used when we need to model a shared but limited access. The resource can be simple, when the discontinuous process that requests for this resource is suspended in case of the resource deficiency. But there is also another type of resources that can be preempted, when another process with more high priority takes the resource ownership from the process with less priority. All this is supported by Aivika.

3.1 Queue Strategies

Before we proceed to more high level modeling constructs, we need to define a *queue strategy*[16] that prescribes how the competitive requests must be prioritized.

In Aivika the queue strategies are expressed in terms of type families, where each queue strategy instance may specify its own queue storage type.

```
class QueueStrategy s where

  data StrategyQueue s :: * -> *

  newStrategyQueue :: s -> Simulation (StrategyQueue s i)
  strategyQueueNull :: StrategyQueue s i -> Event Bool
```

The first function creates a queue by the specified strategy. The second one tests whether the queue is empty.

The `DequeueStrategy` type class defines a strategy applied, when we try to dequeue an item, but there are competitive requests which cannot be fulfilled immediately. Therefore, we have to decide what request has a higher priority when dequeueing.

```
class QueueStrategy s => DequeueStrategy s where
  strategyDequeue :: StrategyQueue s i -> Event i
```

The `EnqueueStrategy` type class defines a strategy applied, when we try to enqueue the item, but there are competitive requests which cannot be fulfilled immediately too. We prioritize the request somehow by adding it to the queue of requests.

```
class DequeueStrategy s => EnqueueStrategy s where
  strategyEnqueue :: StrategyQueue s i -> i -> Event ()
```

There is also a version of the enqueueing strategy that uses priorities.

```
class DequeueStrategy s => PriorityQueueStrategy s p | s -> p where
  strategyEnqueueWithPriority :: StrategyQueue s i -> p -> i -> Event ()
```

There are four predefined queue strategies in Aivika at present:

- FCFS (First Come - First Served), a.k.a. FIFO (First In - First Out);
- LCFS (Last Come - First Served), a.k.a. LIFO (Last In - First Out);
- SIRO (Service in Random Order);
- StaticPriorities (Using Static Priorities).

These strategies are implemented as data types having a single data constructor with the corresponded name.

```
data FCFS = FCFS deriving (Eq, Ord, Show)
data LCFS = LCFS deriving (Eq, Ord, Show)
data SIRO = SIRO deriving (Eq, Ord, Show)
data StaticPriorities = StaticPriorities deriving (Eq, Ord, Show)
```

Each type is an instance of the corresponding queue strategy.

```
instance EnqueueStrategy FCFS
instance EnqueueStrategy LCFS
instance EnqueueStrategy SIRO
instance PriorityQueueStrategy StaticPriorities Double
```

3.2 Resource

A *resource*[8] simulates something to be queued for, for example, the machine.

```
data Resource s
```

Here the parametric type `s` represents a queue strategy. We can use either the predefined strategy or our own custom-made queue strategy. It will work with the both.

The simplest constructor allows us to create a new resource by the specified queue strategy and initial amount.

```
newResource :: QueueStrategy s => s -> Int -> Simulation (Resource s)
```

To acquire the specified resource, we can use the predefined functions like these ones:

```
requestResource :: EnqueueStrategy s => Resource s -> Process ()

requestResourceWithPriority ::
  PriorityQueueStrategy s p => Resource s -> p -> Process ()
```

The both suspend the discontinuous process in case of the resource deficiency until some other simulation activity releases the resource.

```
releaseResourceWithinEvent :: DequeueStrategy s => Resource s -> Event ()
```

There is also a more convenient version of the last function that works within the `Process` computation, but the provided function emphasizes the fact that releasing the resource cannot block the simulation process and this action is performed immediately.

```
releaseResource :: DequeueStrategy s => Resource s -> Process ()
```

We can request for the current available amount of the specified resource as well as request for its capacity and the strategy applied.

```
resourceCount :: Resource s -> Event Int
resourceMaxCount :: Resource s -> Maybe Int
resourceStrategy :: Resource s -> s
```

The second function returns an optional value indicating that the maximum amount could be unspecified when creating the resource.

```
newResourceWithMaxCount ::
  QueueStrategy s => s -> Int -> Maybe Int -> Simulation (Resource s)
```

By default, the maximum possible amount, i.e. the resource capacity, is set equaled to the initial amount specified when calling the first constructor `newResource`.

There are type synonyms for resources that use the predefined queue strategies.

```
type FCFSResource = Resource FCFS
type LCFSResource = Resource LCFS
type SIROResource = Resource SIRO
type PriorityResource = Resource StaticPriorities
```

There are constructors that use these type synonyms. Some of these constructors are used further in the examples.

```
newFCFSResource :: Int -> Simulation FCFSResource
newLCFSResource :: Int -> Simulation LCFSResource
newSIROResource :: Int -> Simulation SIROResource
newPriorityResource :: Int -> Simulation PiriorityResource
```

Finally, there is a helper `usingResource` function that acquires the resource, runs the specified `Process` computation and finally releases the resource regardless of whether the specified process was cancelled or an exception was raised.

```
usingResource :: EnqueueStrategy s => Resource s -> Process a -> Process a
usingResource r m =
  do requestResource r
     finallyProcess m $ releaseResource r
```

We can do the same with the resource using priorities.

```
usingResourceWithPriority ::
  PriorityQueueStrategy s p => Resource s -> p -> Process a -> Process a
usingResourceWithPriority r priority m =
  do requestResourceWithPriority r priority
     finallyProcess m $ releaseResource r
```

The both functions use the `finallyProcess` function that allows executing a finalization part within the computation, whenever an exception might arise or the computation was canceled at all. The functions guarantee that the resource will be released in any case.

3.3 Example: Using Resources

To illustrates how the resources can be used for modeling, let us again take a task from the documentation of SimPy[14].

> Two machines, but sometimes break down. Up time is exponentially distributed with mean 1.0, and repair time is exponentially distributed with mean 0.5. In this example, there is only one repairperson, so the two machines cannot be repaired simultaneously if they are down at the same time.
>
> In addition to finding the long-run proportion of up time, let us also find the long-run proportion of the time that a given machine does not have immediate access to the repairperson when the machine breaks down. Output values should be about 0.6 and 0.67.

In Aivika we can solve this task in the following way.

```
import Control.Monad
import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 1000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation Results
model =
  do -- number of times the machines have broken down
     nRep <- newRef 0

     -- number of breakdowns in which the machine
     -- started repair service right away
     nImmedRep <- newRef 0

     -- total up time for all machines
     totalUpTime <- newRef 0.0

     repairPerson <- newFCFSResource 1

     let machine :: Process ()
         machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)

              -- check the resource availability
              liftEvent $
                do modifyRef nRep (+ 1)
                   n <- resourceCount repairPerson
                   when (n == 1) $
                     modifyRef nImmedRep (+ 1)
```

```
              requestResource repairPerson
              repairTime <-
                randomExponentialProcess meanRepairTime
              releaseResource repairPerson

              machine

     runProcessInStartTime machine
     runProcessInStartTime machine

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

         immedProp :: Event Double
         immedProp =
           do n <- readRef nRep
              nImmed <- readRef nImmedRep
              return $
                fromIntegral nImmed /
                fromIntegral n

     return $
       results
       [resultSource
        "upTimeProp"
        "The long-run proportion of up time (~ 0.6)"
        upTimeProp,
        --
        resultSource
        "immedProp"
        "The proption of time of immediate access (~0.67)"
        immedProp]

main =
  printSimulationResultsInStopTime
  printResultSourceInEnglish
  model specs
```

This is a complete Haskell program, which can be launched with help of the runghc utility or compiled to a native code. It returns the expected results.

```
$ runghc MachRep2.hs
----------

-- simulation time
t = 1000.0

-- The long-run proportion of up time (~ 0.6)
upTimeProp = 0.6089071789353254

-- The proption of time of immediate access (~0.67)
immedProp = 0.6650082918739635
```

Here we use an additional entity, the resource. Actually, there are different resources in Aivika. The resource we have applied is the most simple one, which is optimized for execution.

3.4 Resource Statistics

There is another kind of resources that collect statistics during simulation. Such a resource collects data about its queue size, wait time, contents and utilization.

At first, it resides in another module, because of which we should use a qualified import like this

```
import qualified Simulation.Aivika.Resource as R
```

Since some of these data depend on the starting time, we construct the resource already within the `Event` computation, using the `runEventInStartTime` function if needed.

```
newResource :: QueueStrategy s => s -> Int -> Event (Resource s)

newResourceWithMaxCount ::
  QueueStrategy s => s -> Int -> Maybe Int -> Event (Resource s)
```

In the course of simulation, you can always receive the current state of statistics by functions like this

```
resourceWaitTime :: Resource s -> Event (SamplingStats Double)
```

It returns the resource wait time statistics. Here the `SamplingStats` type corresponds to the statistics based upon Observations. This along with another kind of statistics are described in chapter 4 in more detail.

But in most cases there is no need to request for the statistics explicitly by using such functions. Instead, we will request for these statistics at level of processing the results of simulation as it will be shown in the example from section 3.7.

Finally, we can reset the resource statistics by calling the next function.

```
resetResource :: Resource s -> Event ()
```

Usually, we should call it at some modeling time by using the `enqueueEvent` function to activate the corresponding event for clearing the effects of dirty start, when the model could not be in stable state yet.

In the rest, this resource has almost the same functions that we considered in the previous section 3.2.

3.5 Example: Collecting Resource Statistics

Let us proceed with our model from section 3.3, but rewrite it by using the resource that collects the statistics when simulating.

```
import Control.Monad
import Control.Monad.Trans

import Simulation.Aivika
import qualified Simulation.Aivika.Resource as R

meanUpTime = 1.0
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
```

```
                spcStopTime = 1000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation Results
model =
  do -- number of times the machines have broken down
     nRep <- newRef 0

     -- number of breakdowns in which the machine
     -- started repair service right away
     nImmedRep <- newRef 0

     -- total up time for all machines
     totalUpTime <- newRef 0.0

     repairPerson <- runEventInStartTime $
                     R.newFCFSResource 1

     let machine :: Process ()
         machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)

              -- check the resource availability
              liftEvent $
                do modifyRef nRep (+ 1)
                   n <- R.resourceCount repairPerson
                   when (n == 1) $
                     modifyRef nImmedRep (+ 1)

              R.requestResource repairPerson
              repairTime <-
                randomExponentialProcess meanRepairTime
              R.releaseResource repairPerson

              machine

     runProcessInStartTime machine
     runProcessInStartTime machine

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

         immedProp :: Event Double
         immedProp =
           do n <- readRef nRep
              nImmed <- readRef nImmedRep
              return $
                fromIntegral nImmed /
                fromIntegral n

     return $
       results
       [resultSource
        "upTimeProp"
        "The long-run proportion of up time (~ 0.6)"
```

```
        upTimeProp,
        --
        resultSource
        "immedProp"
        "The proption of time of immediate access (~0.67)"
        immedProp,
        --
        resultSource
        "repairPerson"
        "The repair person"
        repairPerson]

main =
  printSimulationResultsInStopTime
  printResultSourceInEnglish
  model specs
```

This is almost the same as the previous model. Only we import the resource by using the qualified import with letter R.

Now if we will try to launch the simulation then we will see something new, a detailed and verbose dump of the resource state statistics:

```
$ runghc MachRep2.hs
----------

-- simulation time
t = 1000.0

-- The long-run proportion of up time (~ 0.6)
upTimeProp = 0.5868289484150548

-- The proption of time of immediate access (~0.67)
immedProp = 0.6633249791144528

-- The repair person
repairPerson:

  -- the current queue length
  queueCount = 0

  -- the queue length statistics
  queueCountStats = { count = 807, mean = 0.2168308845679987,
    std = 0.412086461880826 54, min = 0 (t = 0.0),
    max = 1 (t = 2.7303659442741184), t in [0.0, 994.4613132444345] }

  -- the total wait time
  totalWaitTime = 215.62992621944437

  -- the wait time
  waitTime = { count = 1197, mean = 0.1801419600830781,
    std = 0.39685050303019626, min = 0.0, max = 3.6879403657935086 }

  -- the current available count
  count = 0

  -- the available count statistics
  countStats = { count = 1588, mean = 0.3903529011661833,
    std = 0.48782938996879555, min = 0 (t = 0.6356902271276655),
    max = 1 (t = 0.0), t in [0.0, 999.4165355910017] }

  -- the current utilisation count
  utilisationCount = 1
```

```
-- the utilisation count statistics
utilisationCountStats = { count = 2394, mean = 0.6096470988338166,
  std = 0.48782938996879555, min = 0 (t = 0.0),
  max = 1 (t = 0.6356902271276655), t in [0.0, 999.4165355910017] }
```

The dump is quite self-explanatory. It shows the queue count, its statistics, the wait time and the corresponding statistics as well as the similar information for the resource amount and utilisation count. In chapter 4 you will know more about types that are used for gathering the statistics in Aivika. Here we see their `String` representation.

Earlier we saw different charts plotted within the simulation experiments. As you will see further, we can also use the resource properties in the series queries when plotting the charts. Moreover, the point is that we can make such series queries to arbitrary properties of those objects that support the protocol of the `Results` type, that is, those ones that can be returned within the simulation results, for example, the queues and servers to name a few too.

3.6 Referencing to Properties

As we saw, the resource can have additional fields with statistical data. We can refer to these data either by direct calling the accessor functions, or by specifying what namely results we want to use. Here we will consider the second approach.

So, there is the following module that contains helper functions to work with the simulation results.

```
module Simulation.Aivika.Results.Transform
```

Earlier we saw that we can specify what series we want to use when plotting the chart. The same way, we can specify that we want to use a series corresponding to the resource's queue count:

```
resourceCount :: Resource -> ResultTransform
```

The `ResultTransform` represents the series, but the `Resource` type is not those resource types we saw before. Within the considered module, this is a separate type, which has one data constructor that allows creating a new resource of this kind by the specified series.

```
newtype Resource = Resource ResultTransform
```

As you can remember, we refer to the series by its name. So, we refer to some resource by its name and then call the `Resource` data constructor to create a new value of the `Resource` data type. Then we can access to additional series by functions like `resourceCount`. The resource itself is a compound series object, which cannot be plotted directly on the chart as it consists of many series.

We can write something like this

```
r = Resource $ resultByName "repairPerson"
s = resourceCount r
```

Below we will plot the chart for the resource utilization. For that, we will need the following function that returns the corresponding series.

```
resourceUtilisationCount :: Resource -> ResultTransform
```

3.7 Example: Charts for Resource Properties

Now we will illustrate how we can plot the chart for the resource utilisation in our previous example. As before, we divide the code into different files, where the first file will correspond to the model itself, the second file will describe the experiment and another file will launch the simulation by using one of the charting back-ends.

3.7.1 Returning Results from Model

Here we just rewrite slightly the code putting the simulation model in a separate module. There is no essential difference from the code provided in section 3.5.

```
module Model (model) where

import Control.Monad
import Control.Monad.Trans

import Simulation.Aivika
import qualified Simulation.Aivika.Resource as R

meanUpTime = 1.0
meanRepairTime = 0.5

model :: Simulation Results
model =
  do -- number of times the machines have broken down
     nRep <- newRef 0

     -- number of breakdowns in which the machine
     -- started repair service right away
     nImmedRep <- newRef 0

     -- total up time for all machines
     totalUpTime <- newRef 0.0

     repairPerson <- runEventInStartTime $
                     R.newFCFSResource 1

     let machine :: Process ()
         machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)

              -- check the resource availability
              liftEvent $
                do modifyRef nRep (+ 1)
                   n <- R.resourceCount repairPerson
                   when (n == 1) $
                     modifyRef nImmedRep (+ 1)

              R.requestResource repairPerson
              repairTime <-
                randomExponentialProcess meanRepairTime
              R.releaseResource repairPerson

              machine
```

```
    runProcessInStartTime machine
    runProcessInStartTime machine

    let upTimeProp =
          do x <- readRef totalUpTime
             y <- liftDynamics time
             return $ x / (2 * y)

        immedProp :: Event Double
        immedProp =
          do n <- readRef nRep
             nImmed <- readRef nImmedRep
             return $
               fromIntegral nImmed /
               fromIntegral n

    return $
      results
      [resultSource
       "upTimeProp"
       "The long-run proportion of up time (~ 0.6)"
       upTimeProp,
       --
       resultSource
       "immedProp"
       "The proption of time of immediate access (~0.67)"
       immedProp,
       --
       resultSource
       "repairPerson"
       "The repair person"
       repairPerson]
```

3.7.2 Experiment Definition

Now the most exciting thing follows. We define the simulation experiment in declarative manner. We want to plot the trend for the resource utilization. The chart will also show the confidence intervals by rule 3-sigma.

```
module Experiment (experiment, generators) where

import Data.Monoid

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

import qualified Simulation.Aivika.Results.Transform as T

specs = Specs { spcStartTime = 0,
                spcStopTime = 1000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1000,
    experimentTitle = "MachRep2",
```

```
    experimentDescription = "Example: Charts for Resource Properties" }

upTimeProp = resultByName "upTimeProp"
immedProp = resultByName "immedProp"

repairPerson = T.Resource $ resultByName "repairPerson"
repairPersonUtil = T.resourceUtilisationCount repairPerson

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "Resource Utilization",
     deviationChartLeftYSeries = repairPersonUtil },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "Resource Utilization Stats.",
     finalStatsSeries = repairPersonUtil }]
```

Also we specify that the resulting Web page will show the statistics summary for the resource utilization. The number of simulation runs in 1000.

Note how we gain access to the resource property. We treat the specified result source as a resource and then safely call the corresponding accessor.

3.7.3 Charting

Here you can choose one of the charting back-ends. The code is absolutely the same as it was in section 1.4.3.

3.7.4 Running Simulation Experiment

The simulation experiment consisting of 1000 runs, compiled with option `-O2` and run with options `+RTS -N`, lasted for about 3 seconds on my computer having the 2-core processor with hyper-threading, i.e. with 4 virtual cores. If you have an 8-core or 36-core processor, the speed of simulation will even be more fast.

Table 3.1: The repair person utilisation statistics by data collected in final time points.

mean	0.5950000000000005
deviation	0.4911376754192414
minimum	0.0
maximum	1.0
count	1000

You can find the corresponding chart on figure 3.1 that shows the trend and confidence intervals by rule 3-sigma.

3.8 Resource Preemption

The `Process` computation supports a special very useful feature, which is called a *preemption*. To be not formally strict, the preemption means that some discontinuous process can be temporarily interrupted so that it could resume its execution later. It affects the statistics, affects the resource ownership etc.

In Aivika the process preemption is hidden from direct using. Instead, there are some data types that provide with high-level interfaces that already use the preemption under the hood.

One of such data types is a special kind of resource considered in this section. As before, there are two similar modules that provide with the resource type: a simple one but optimized for execution and a more full version of the resource that updates its statistics. For simplicity, we will consider the latter.

This resource resides in its own module and has the predefined queue strategy. So, there is no need to specify the strategy as we did for more simple resources in the previous sections.

```
module Simulation.Aivika.Resource.Preemption

data Resource
```

As before, there are two constructing functions that allow us to create a new resource. There is the mandatory initial amount and an optional capacity:

```
newResource :: Int -> Event Resource
newResourceWithMaxCount :: Int -> Maybe Int -> Event Resource
```

To acquire the resource, the discontinuous process has to supply with its priority. If the resource already belongs to another process with less priority, then the old process is preempted and the current process takes an ownership of the resource. Otherwise, if the old process has the same priority or higher, then the current process waits for releasing the resource. Note that the less value means a higher priority.

```
requestResourceWithPriority :: Resource -> Double -> Process ()
```

To release the resource, we call the next function. If some process was preempted before, then that process resumes its execution from the point where it was preempted.

```
releaseResource :: Resource -> Process ()
```

Unfortunately, the current implementation of the resource preemption has a limitation. The resource can be released only by that process that acquired the resource before. If you need a more complex behavior then you probably should consider using the GPSS-like DSL considered in chapter 10.

To reset the resource statistics at some modeling time, we can call the following function.

```
resetResource :: Resource -> Event ()
```

In section 6.8 you can find an example of using the resource preemption, but we need to introduce some new concepts considered further.

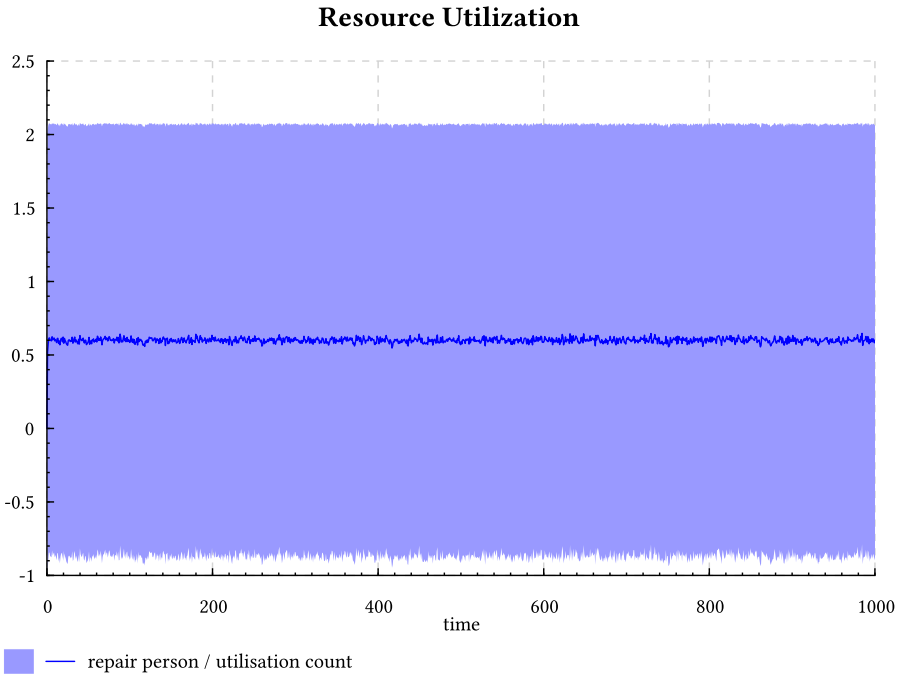


Figure 3.1: The deviation chart for the resource utilization.

Chapter 4

Statistics

Accumulating statistics is an important part of simulation. Aivika uses the approach, where the statistics summary is treated as an immutable data structure, which simplifies programming and makes the simulation more safe and robust.

There are two different types of statistics that we can collect. The first one is based upon observations, while the latter is based on time-dependent samples.

4.1 Statistics based upon Observations

The `SamplingStats` data type is used for accumulating the statistics based upon observations. An example is the queue wait time.

```
data SamplingStats a

class Num a => SamplingData a where

  emptySamplingStats :: SamplingStats a
  addSamplingStats :: a -> SamplingStats a -> SamplingStats a
  combineSamplingStats :: SamplingStats a -> SamplingStats a -> SamplingStats a
```

The first function returns an empty statistics. The `addSamplingStats` function takes a new sample value and some statistics, but then returns a new accumulating statistics. The third function takes two statistics and returns the combined statistics.

The are two important instances that allow creating numerical statistics.

```
instance SamplingData Int
instance SamplingData Double
```

The usual mistake of novices is when they try to use rather a heavy-weight `Var` type for collecting statistics. Nevertheless, it is recommended to use the `Ref` reference for iterative updating the light-weight `SamplingStats` values, which is a more efficient and more simple approach.

By the specified `SamplingStats` value, we can receive the statistics summary that includes the number of observations, minimum, maximum, average value, average square value, variation and deviation, respectively.

```
samplingStatsCount :: SamplingStats a -> Int
samplingStatsMin   :: SamplingStats a -> a
samplingStatsMax   :: SamplingStats a -> a
samplingStatsMean  :: SamplingStats a -> Double
samplingStatsMean2 :: SamplingStats a -> Double
samplingStatsVariance :: SamplingStats a -> Double
samplingStatsDeviation :: SamplingStats a -> Double
```

Finally, the `SamplingStats` statistics can be returned as a `ResultSource` from the simulation model.

4.2 Statistics for Time Persistent Variables

The `TimingStats` statistics is very similar to `SamplingStats`, but only the former allows collecting samples bound to time points. The corresponding random variable is sometimes called *time persistent*. For example, the queue length is an example of such a time persistent variable.

```
data TimingStats a

class Num a => TimingData a where

  emptyTimingStats :: TimingStats a
  addTimingStats :: Double -> a -> TimingStats a -> TimingStats a
```

As before, we can create an empty statistics, but when updating the statistics, we have also to specify the corresponding time in the first argument passing it to the `addTimingStats` function. Moreover, there is no analog of the combining function for this type of statistics.

By the way, namely because of need to specify the time, the resource constructor from section 3.4 returns an action within the `Event` computation, not within the `Simulation` one as we might expect. We just initialize some resource statistics precisely at time of creating the resource.

The `TimingStats` returns more data. At first, it returns the same set of properties: the number of samples, minimum, maximum, average value, average square value, variance and deviation.

```
timingStatsCount :: TimingStats a -> Int
timingStatsMin   :: TimingStats a -> a
timingStatsMax   :: TimingStats a -> a
timingStatsMean  :: TimingData a => TimingStats a -> Double
timingStatsMean2 :: TimingData a => TimingStats a -> Double
timingStatsVariance :: TimingData a => TimingStats a -> Double
timingStatsDeviation :: TimingData a => TimingStats a -> Double
```

Here the number of samples has already a quite conditional meaning unlike the previous type of statistics based upon observations.

Additionally, the `TimingStats` value returns the last accumulated value, time at which the minimum is attained, time as which the maximum is attained, start time of sampling, last time of sampling, sum of values and sum of square values, respectively.

```
timingStatsLast :: TimingStats a -> a
timingStatsMinTime :: TimingStats a -> Double
timingStatsMaxTime :: TimingStats a -> Double
```

```
timingStatsStartTime :: TimingStats a -> Double
timingStatsLastTime :: TimingStats a -> Double
timingStatsSum :: TimingStats a -> Double
timingStatsSum2 :: TimingStats a -> Double
```

The are also two important instances that allow creating numerical statistics of this kind.

```
instance TimingData Int
instance TimingData Double
```

As before, the `TimingStats` value can be returned from the model as a `ResultSource`.

We can see that only the first two moments are calculated by these two types `SamplingStats` and `TimingStats`, but it should cover the most of use cases. If you need a more detailed statistical analysis, then you probably need to gather and process the statistics themselves.

Chapter 5

Signals and Tasks

In Aivika there is a signalling mechanism which is based on ideas of the .NET `IObservable` interface. The signal is something that notifies its listeners about changes, events and so on. The listener may subscribe to receiving the signal values, but then unsubscribe.

We can also define a task to represent the discontinuous processes running in foreground. By triggering the corresponding signal, the task will notify us that the process has finished and returned the result.

5.1 Signaling

The following monoid represents a signal that notifies about occurring some condition.

```
data Signal a =
  Signal { handleSignal :: (a -> Event ()) -> Event DisposableEvent }
```

The `handleSignal` function takes a signal and its handler, subscribes the handler for receiving the signal values and then returns a nested computation of the `DisposableEvent` type that being applied unsubscribes the specified handler from receiving the signal.

The act of unsubscribing from the signal occurs in a time. Therefore, the nested computation returned has actually type `Event ()` hidden under the facade of the convenient type name.

```
disposeEvent :: DisposableEvent -> Event ()
```

If we are not going to unsubscribe at all, then we can ignore the nested computation.

```
handleSignal_ :: Signal a -> (a -> Event ()) -> Event ()
```

We can treat the signals in a functional way, transforming, or merging, or filtering them with help of combinators.

```
instance Monoid (Signal a)
instance Functor Signal

filterSignal :: (a -> Bool) -> Signal a -> Signal a
```

The `Ref` reference and `Var` variable provide signals that notify about changing their state.

```
refChanged :: Ref a -> Signal a
varChanged :: Var a -> Signal a
```

We can create an origin of the signal manually. Distinguishing the origin from the signal allows us to publish the signal with help of a pure function. But we have to trigger the signal within a computation synchronized with the event queue, though.

```
data SignalSource a

newSignalSource :: Simulation (SignalSource a)

publishSignal :: SignalSource a -> Signal a
triggerSignal :: SignalSource a -> a -> Event ()
```

5.2 Tasks

There is a link between the signals and discontinuous processes, which is expressed by the following function that suspends the current process until a signal comes.

```
processAwait :: Signal a -> Process a
```

Only one signal value is expected, but then the process automatically unsubscribes from the specified signal.

In Aivika there is an opposite transformation from the `Process` computation to a `Signal` value, but it is a little bit complicated as the process can be actually canceled, or an `IO` exception can be raised within the simulation. The corresponding transformation is defined with help of the `Task` type.

```
runTask :: Process a -> Event (Task a)
```

Here we run the specified process in background and immediately return the corresponding task within the `Event` computation. Later we can request for the result of the underlying `Process` computation, whether it was finished successfully, or an `IO` exception had occurred, or the computation was cancelled. Please refer to the Aivika documentation for detail.

Using signals, the mentioned earlier function `timeoutProcess` is implemented. It allows us to run a sub-process within the specified time-out. The function creates an internal signal source. The launched sub-process is trying to compute the result and in case of success it notifies the parent process, triggering the corresponding signal. Only it is worth noting again that the `timeoutProcess` function is very slow.

The signals are also used in the `memoProcess` function. It takes the specified process and returns a new resulting process that will always return the same value within the current simulation run regardless of that how many times that process will be started. The value is calculated only once for each simulation run but all other process instances await the signal for receiving the result.

Finally, the signals are extensively used in simulation experiments. Namely by triggering the corresponding signal, the chart plotters and other renderers receive the information that the specified result source has changed.

5.3 Composites

You could notice that the act of subscribing to receiving the signal values returns the `DisposableEvent` value within the `Event` computation. This is so a common pattern, because of which a separate monad was introduced.

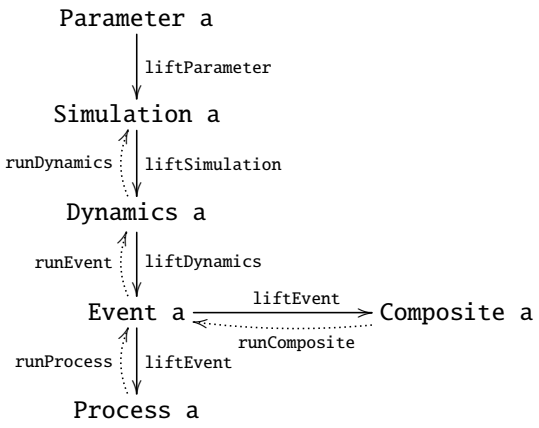
```
data Composite a

runComposite :: Composite a -> DisposableEvent -> Event (a, DisposableEvent)
runComposite_ :: Composite a -> Event a
```

The first run function allows building a composite that can be then destroyed by calling the corresponding `DisposableEvent` computation.

When constructing the composite, we can add our own future destroying action. This is the main feature of the considered computation.

```
disposableComposite :: DisposableEvent -> Composite ()
```

The `Composite` computation is based on the `Event` computation. This is essentially the `Event` computation, but which remembers all its `DisposableEvent` actions, which can be then applied when destroying the composite. The `Composite` computation is often used together with signals.



Chapter 6

Queue Network

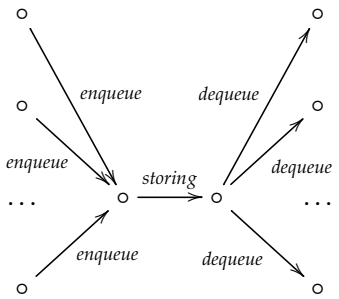
6.1 Queues

Sometimes we need a location in the network where entities wait for service[11]. They are modeled in Aivika by the bounded and unbounded queues.

For brevity, only the bounded queue is considered here as this is a more difficult case.

```
data Queue si sm so a
```

It represents a queue using the specified strategies for enqueueing (input), `si`, internal storing (in memory), `sm`, and dequeueing (output), `so`, where the parametric type a denotes the type of items stored in the queue.



There are type synonyms for the most important use cases:

```
type FCFSQueue a = Queue FCFS FCFS FCFS a
type LCFSQueue a = Queue FCFS LCFS FCFS a
type SIROQueue a = Queue FCFS SIRO FCFS a
type PriorityQueue a = Queue FCFS StaticPriorities FCFS a
```

These synonyms use the FIFO strategy both for the enqueue and dequeue requests, which seems to be reasonable for many cases. However, we can define a bounded queue that would process all pending dequeue requests, for example, in random (SIRO) or opposite (LIFO) order in case of enqueueing an item in the empty queue. Here we operate on the same queue strategies that we introduced in section 3.1.

There are different functions that allow creating a new empty queue by the specified capacity.

```
newQueue :: (QueueStrategy si, QueueStrategy sm, QueueStrategy so)
            => si -> sm -> so -> Int -> Event (Queue si sm so a)

newFCFSQueue :: Int -> Event (FCFSQueue a)
newLCFSQueue :: Int -> Event (LCFSQueue a)
newSIROQueue :: Int -> Event (SIROQueue a)
newPriorityQueue :: Int -> Event (PriorityQueue a)
```

The queue is created within the `Event` computation as we have to know the current simulation time to start gathering the timing statistics for the queue size. The statistics is initiated at time of invoking the computation.

There are different enqueue functions. The most simple one is provided below.

```
enqueue :: (EnqueueStrategy si, EnqueueStrategy sm, DequeueStrategy so)
           => Queue si sm so a -> a -> Process ()
```

It suspends the process if the bounded queue is full. Therefore, this action is returned as the `Process` computation.

Also we can try to enqueue the specified item and if the queue is full then the item is counted as lost.

```
enqueueOrLost :: (EnqueueStrategy sm, DequeueStrategy so)
                 => Queue si sm so a -> a -> Event Bool
```

This action cannot already suspend the simulation activity and hence it returns the `Event` computation of a flag indicating whether the item was successfully stored in the queue.

The simplest dequeue operation suspends the process while the queue is empty. The result is the `Process` computation again.

```
dequeue :: (DequeueStrategy si, DequeueStrategy sm, EnqueueStrategy so)
           => Queue si sm so a -> Process a
```

Here the very type signatures specify whether the corresponding action may suspend the simulation activity, or the action is performed immediately.

There are similar enqueue and dequeue functions that allow specifying the priorities if the corresponding queue strategy supports them.

The queue has a lot of counters that are updated during simulation. Actually, these counters are what we are mostly interested in.

For example, we can request the queue for its size and wait time statistics. We considered statistics data types in chapter 4.

```
queueCountStats :: Queue si sm so a -> Event (TimingStats Int)
queueWaitTime :: Queue si sm so a -> Event (SamplingStats Double)
```

Here `SamplingStats` is a relatively light-weight and immutable data type that comprises the statistics summary collected for the queue's wait time. It returns the average value, variance, deviation, minimum, maximum and the number of samples. The `TimingStats` provides additional information about the times at which the minimum and maximum values were gained. Also the timing statistics takes into account the modeling time at which data are gathered.

To reset the queue statistics at the current modeling time, we can call the following function:

```
resetQueue :: Queue si sm so a -> Event ()
```

We can request for the queue properties in the simulation experiment like that how we requested for the resource properties in section 3.6.

```
import qualified Simulation.Aivika.Results.Transform as T

q = T.Queue $ resultByName "someQueue"
k = T.tr $ T.queueWaitTime q
```

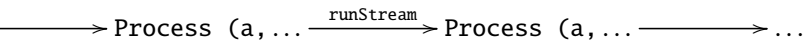
Below is shown how the queues can be processed using more high level computations that operate on the stream of data.

6.2 Stream

Many things become significantly more simple for reasoning and understanding after we introduce a concept of *infinite stream* of data that come with some delays in the modeling time.

```
newtype Stream a = Cons { runStream :: Process (a, Stream a) }
```

This is a kind of the famous *cons-cell*, where the cell is already returned within the `Process` computation. It means that the stream data can be distributed in the modeling time and there can be time gaps between arrivals of sequential data.



The stream represents sequential data. For example, if you need the processing of parallel transacts then you should probably consider using the GPSS-like DSL, which is also supported by Aivika. It is described in section 10.

The streams themselves are well-known in the functional programming for a long time[1]. It is obvious that we can map, filter, and transform the streams.

Now it is more interesting what new properties we can gain by introducing the `Process` computation in the cons-cell definition. At least, the `Stream` type is a monoid.

Passivating the underlying process forever[1], we receive a stream that never returns data.

```
emptyStream :: Stream a
```

Moreover, we can merge two streams applying the `FCFS` strategy when enqueueing the input data.

```
mergeStreams :: Stream a -> Stream a -> Stream a
```

Actually, the latter is a partial case of more general functions that allow concatenating the streams like a *multiplexor*.

[1]The underlying process can still be canceled, though.

```
concatStreams :: [Stream a] -> Stream a
concatStreams = concatQueuedStreams FCFS

concatQueuedStreams :: EnqueueStrategy s => s -> [Stream a] -> Stream a

concatPriorityStreams ::
  PriorityQueueStrategy s p => s -> [Stream (p, a)] -> Stream a
```

The functions use the resources to concatenate different infinite streams of data.

There is an opposite ability to split the input stream into the specified number of output streams like a *demultiplexor*. We have to do it to model a parallel work of services.

```
splitStream :: Int -> Stream a -> Simulation [Stream a]
splitStream = splitStreamQueueing FCFS

splitStreamQueueing ::
  EnqueueStrategy s
  => s -> Int -> Stream a -> Simulation [Stream a]

splitStreamPrioritising ::
  PriorityQueueStrategy s p
  => s -> [Stream p] -> Stream a -> Simulation [Stream a]
```

An implementation of the second function is provided below for demonstrating the approach. Only we need an auxiliary function that creates a new stream as a result of the repetitive execution of some process.

```
repeatProcess :: Process a -> Stream a
```

Here is the `splitStreamQueueing` function itself:

```
splitStreamQueueing s n x =
  do ref <- liftIO $ newIORef x
     res <- newResource s 1
     let reader =
           usingResource res $
           do p <- liftIO $ readIORef ref
              (a, xs) <- runStream p
              liftIO $ writeIORef ref xs
              return a
     return $ map (\i -> repeatProcess reader) [1..n]
```

A key idea is that many simulation models can be defined as a network of the `Stream` computations.

Such a network must have external input streams, usually random streams like these ones.

```
randomUniformStream :: Double -> Double -> Stream (Arrival Double)
randomNormalStream :: Double -> Double -> Stream (Arrival Double)
randomExponentialStream :: Double -> Stream (Arrival Double)
randomErlangStream :: Double -> Int -> Stream (Arrival Double)
randomPoissonStream :: Double -> Stream (Arrival Int)
randomBinomialStream :: Double -> Int -> Stream (Arrival Int)
```

Here a value of type `Arrival a` contains the modeling time at which the external event has arrived, the event itself of type `a` and the delay time which has passed from the time of arriving the previous event.

To process the input stream in parallel, we split the input with help of the `splitStream` function, process new streams in parallel and then concatenate the intermediate results into one output stream using the `concatStreams` function. Later will be provided the `processorParallel` function that does namely this.

To process the specified stream sequentially by some servers, we need a helper function that would read one more data item in advance, playing a role of the intermediate buffer between the servers.

```
prefetchStream :: Stream a -> Stream a
```

Now we need the moving force that would run the whole network of streams.

```
sinkStream :: Stream a -> Process ()
```

It infinitely reads data from the specified stream. This is like a terminator in GPSS.

6.3 Passive Streams and Active Signals

At least, there are two different types of data sources: streams and signals.

The data can be requested explicitly. If we don't request them then they don't come. This is what the `Stream` computation defines. Only the random streams considered in the previous section check that the data are requested permanently and in order. If the random streams are requested wrong then there will be a run-time error. This is something like a sanity check that the model has no logical errors. But the streams themselves, in general, have no such a check.

Therefore, you should be very careful, when using the `Stream` computation. The stream must be permanently requested. You should either use the `sinkStream` terminator, or put the items in the queue, or exclude the items from the simulation, for example, if the queue is full.

Probably, the GPSS-like DSL described in chapter 10 is more safe and easy-to-use as it has no these constraints. Also that DSL is very powerful. Only it still uses streams when defining generators.

However, the streams can be useful for modeling many *queue networks*. For example, it is natural to represent the arrival of orders as the `Stream` computation.

Unlike the streams, the data in signals arrive regardless of that we process them or not. We can subscribe to handing the `Signal` values and then we will receive these values. If we don't subscribe, the signal will trigger its values anyway. Therefore, we can say that the `Signal` computation is active, while the `Stream` computation is passive.

For example, we can model WiFi signals as the `Signal` computation. This computation can be useful in other cases too.

6.4 Processor

Having a stream of data, it would be natural to operate on its transformation which we will call a *processor*:

```
newtype Processor a b = Processor { runProcessor :: Stream a -> Stream b }
```

Here the choice of this name was not arbitrary [4]. This type seems to be an `Arrow`, but the arrow can be interpreted as some kind of processor. Only the code will probably be slow if you will decide to use the *proc*-notation as the `Arrow` instance for this particular type is not very optimal.

We can construct the processors directly from the streams. Omitting the obvious cases, we consider only the most important ones.

A new processor can be created by the specified handling function producing the `Process` computation.

```
arrProcessor :: (a -> Process b) -> Processor a b
```

Also we can use an accumulator to save an intermediate state of the processor. When processing the input stream and generating an output one, we can update the state.

```
accumProcessor :: (acc -> a -> Process (acc, b)) -> acc -> Processor a b
```

An arbitrary number of processors can be united to work in parallel using the default `FCFS` queue strategy:

```
processorParallel :: [Processor a b] -> Processor a b
```

Its implementation is based on using the multiplexing an demultiplexing functions considered before. We split the input stream, process the intermediated streams in parallel and then concatenate the resulting streams into one output steam.

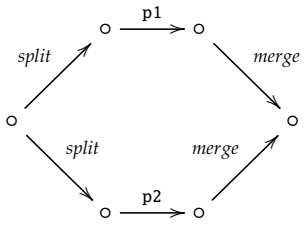
There are other versions of the `processParallel` function, where we can specify the queue strategies and priorities if required.

To create a sequence of autonomously working processors, we can use the prefetching function considered above too:

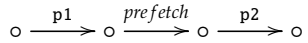
```
prefetchProcessor :: Processor a a
prefetchProcessor = Processor prefetchStream
```

For example, having two complementing processors `p1` and `p2`, we can create two new processors, where the first one implies a parallel work, while another implies a sequential processing:

```
pPar = processorParallel [p1, p2]
```




```
pSeq = p1 >>> prefetchProcessor >>> p2
```



Basing on this approach, we can model in Haskell quite complex queue networks in an easy-to-use high-level declarative manner.

Regarding the queues themselves, we can model them using rather general-purpose helper processors like this one:

```
queueProcessor :: (a -> Process ())
                  -- ^ enqueue the input item
                  -> Process b
                  -- ^ dequeue an output item
                  -> Processor a b
                  -- ^ the buffering processor
```

An idea is that there is a plenty of cases how the queues could be united in the network. When enqueueing, we can either wait while the queue is full, or we can count such an item as lost. We can use the priorities for the `Process` computations that enqueue or dequeue. Moreover, different processes can enqueue and dequeue simultaneously.

Therefore, the author of Aivika decided to introduce such general-purpose helper functions for modeling the queues, where the details of how the queues are simulated can be shortly described with help of combinators like `enqueue`, `enqueueOrLost` and `dequeue` stated above. The examples are included in the Aivika distribution.

Unfortunately, the `Processor` type is not `ArrowLoop` by the same reason why the `Process` monad is not `MonadFix` — continuations. Nevertheless, we can model the queue networks with loopbacks using the intermediate queues to delay the stream. One of the possible functions is provided below.

```
queueProcessorLoopSeq ::
  (a -> Process ())
  -- ^ enqueue the input item
  -> Process c
  -- ^ dequeue an item for the further processing
  -> Processor c (Either e b)
  -- ^ process and then decide what values of type @e@
  -- should be processed in the loop (condition)
  -> Processor e a
  -- ^ process in the loop and then return a value
  -- of type @a@ to the queue again (loop body)
  -> Processor a b
  -- ^ the buffering processor
```

An example model that would use the streams and queue processors is provided further in section 6.7.

Probably, here the reader will find the GPSS-like DSL described in chapter 10 more convenient for using in his/her models. Aivika naturally supports this DSL too. It is also based on the `Process` computation.

Using the processors, we can model a complicated enough behavior. For example, we can model the Round-Robbin strategy[16] of the processing.

```
roundRobbinProcessor :: Processor (Process Double, Process a) a
```

It tries to perform a task within the specified timeout. If the task times out, then it is canceled and returned to the processor again; otherwise, the

successful result is redirected to output. The timeout and task are passed in to the processor from the input stream.

Both the processors and streams allow modeling the process-oriented simulation on a higher level in a way somewhere similar to that one which is described in book [11] by A. Alan B. Pritsker and Jean J. O'Reilly.

At the same time, all computations are well integrated in Aivika and we can combine different approaches within the same model, for example, combining the process-oriented simulation with agent-based modeling.

6.5 Server

In Aivika there is a `Server` data type that allows modeling a working place and that gathers its statistics during simulation.

```
data Server s a b

newServer :: (a -> Process b) -> Simulation (Server () a b)
newStateServer :: (s -> a -> Process (s, b)) -> s -> Simulation (Server s a b)
```

To create a server, we provide a handling function that takes the input, process it and generates an output within the `Process` computation. The handling function may use an accumulator to save the server state when processing.

To involve the server in simulation, we can use its processor that performs a service and updates the internal counters.

```
serverProcessor :: Server s a b -> Processor a b
```

For example, when preparing the simulation results to output, we can request for the statistics of the time spent by the server while processing the tasks.

```
serverProcessingTime :: Server s a b -> Event (SamplingStats Double)
```

There is one subtle thing. Each time we use the `serverProcessor` function, we actually create a new processor that refers to the same server and hence updates the same statistics counters. It can be useful if we are going to gather the statistics for a group of servers working in parallel, although the best practice would be to use the `serverProcessor` function only once per each server.

To reset the server statistics at the current modeling time, we can call the following function:

```
resetServer :: Server s a b -> Event ()
```

We can request for the server properties in the simulation experiment like that how we requested for the resource properties in section 3.6.

```
import qualified Simulation.Aivika.Results.Transform as T

s = T.Server $ resultByName "someServer"
k = T.tr $ T.serverProcessingTime s
```

6.6 Measuring Processing Time

To measure the processing time of orders, a simple object called `ArrivalTimer` is often used in the examples.

```
data ArrivalTimer

newArrivalTimer :: Simulation ArrivalTimer

arrivalTimerProcessor :: ArrivalTimer -> Processor (Arrival a) (Arrival a)
arrivalProcessingTime :: ArrivalTimer -> Event (SamplingStats Double)
```

Each time you apply the `arrivalTimerProcessor` function, the corresponding internal counter is updated to measure the processing time of arrivals that contain the exact time of entering the simulation model. Then this counter is returned by the `arrivalProcessingTime` function as the statistics summary.

But usually, the processing time is requested for within simulation experiments by the code like this:

```
import qualified Simulation.Aivika.Results.Transform as T

t = T.ArrivalTimer $ resultByName "someTimer"
k = T.tr $ T.arrivalProcessingTime t
```

6.7 Example: Queue Network

To illustrate how the streams and processors can be used for modeling, let us consider a model [11, 16] of inspection and adjustment stations on a production line. This is a model of the workflow with a loop. Also there are two unbounded queues.

> Assembled television sets move through a series of testing stations in the final stage of their production. At the last of these stations, the vertical control setting on the TV sets is tested. If the setting is found to be functioning improperly, the offending set is routed to an adjustment station where the setting is adjusted. After adjustment, the television set is sent back to the last inspection station where the setting is again inspected. Television sets passing the final inspection phase, whether for the first time of after one or more routings through the adjustment station, are routed to a packing area.
>
> The time between arrivals of television sets to the final inspection station is uniformly distributed between 3.5 and 7.5 minutes. Two inspectors work side-by-side at the final inspection station. The time required to inspect a set is uniformly distributed between 6 and 12 minutes. On the average, 85 percent of the sets are routed to the adjustment station which is manned by a single worker. Adjustment of the vertical control setting requires between 20 and 40 minutes, uniformly distributed.
>
> The inspection station and adjustor are to be simulated for 480 minutes to estimate the time to process television sets through the final production stage and to determine the utilization of the inspectors and the adjustors.

6.7.1 Returning Results from Model

Below is provided the model defined in a quite declarative and straightforward way.

```
module Model (model) where

import Prelude hiding (id, (.))

import Control.Monad
import Control.Monad.Trans
import Control.Arrow
import Control.Category (id, (.))

import Simulation.Aivika
import Simulation.Aivika.Queue.Infinite

-- the minimum delay of arriving the next TV set
minArrivalDelay = 3.5

-- the maximum delay of arriving the next TV set
maxArrivalDelay = 7.5

-- the minimum time to inspect the TV set
minInspectionTime = 6

-- the maximum time to inspect the TV set
maxInspectionTime = 12

-- the probability of passing the inspection phase
inspectionPassingProb = 0.85

-- how many are inspection stations?
inspectionStationCount = 2

-- the minimum time to adjust an improper TV set
minAdjustmentTime = 20

-- the maximum time to adjust an improper TV set
maxAdjustmentTime = 40

-- how many are adjustment stations?
adjustmentStationCount = 1

-- create an inspection station (server)
newInspectionStation =
  newServer $ \a ->
  do holdProcess =<<
       (liftParameter $
        randomUniform minInspectionTime maxInspectionTime)
     passed <-
       liftParameter $
       randomTrue inspectionPassingProb
     if passed
       then return $ Right a
       else return $ Left a

-- create an adjustment station (server)
newAdjustmentStation =
  newServer $ \a ->
  do holdProcess =<<
       (liftParameter $
```

```
        randomUniform minAdjustmentTime maxAdjustmentTime)
     return a

model :: Simulation Results
model = mdo
  -- to count the arrived TV sets for inspecting and adjusting
  inputArrivalTimer <- newArrivalTimer
  -- it will gather the statistics of the processing time
  outputArrivalTimer <- newArrivalTimer
  -- define a stream of input events
  let inputStream =
        randomUniformStream minArrivalDelay maxArrivalDelay
  -- create a queue before the inspection stations
  inspectionQueue <-
    runEventInStartTime newFCFSQueue
  -- create a queue before the adjustment stations
  adjustmentQueue <-
    runEventInStartTime newFCFSQueue
  -- create the inspection stations (servers)
  inspectionStations <-
    forM [1 .. inspectionStationCount] $ \_ ->
    newInspectionStation
  -- create the adjustment stations (servers)
  adjustmentStations <-
    forM [1 .. adjustmentStationCount] $ \_ ->
    newAdjustmentStation
  -- a processor loop for the inspection stations' queue
  let inspectionQueueProcessorLoop =
        queueProcessorLoopSeq
        (liftEvent . enqueue inspectionQueue)
        (dequeue inspectionQueue)
        inspectionProcessor
        (adjustmentQueueProcessor >>> adjustmentProcessor)
  -- a processor for the adjustment stations' queue
  let adjustmentQueueProcessor =
        queueProcessor
        (liftEvent . enqueue adjustmentQueue)
        (dequeue adjustmentQueue)
  -- a parallel work of the inspection stations
  let inspectionProcessor =
        processorParallel (map serverProcessor inspectionStations)
  -- a parallel work of the adjustment stations
  let adjustmentProcessor =
        processorParallel (map serverProcessor adjustmentStations)
  -- the entire processor from input to output
  let entireProcessor =
        arrivalTimerProcessor inputArrivalTimer >>>
        inspectionQueueProcessorLoop >>>
        arrivalTimerProcessor outputArrivalTimer
  -- start simulating the model
  runProcessInStartTime $
    sinkStream $ runProcessor entireProcessor inputStream
  -- return the simulation results in start time
  return $
    results
    [resultSource
     "inspectionQueue" "the inspection queue"
     inspectionQueue,
     --
     resultSource
     "adjustmentQueue" "the adjustment queue"
     adjustmentQueue,
```

```
    --
    resultSource
    "inputArrivalTimer" "the input arrival timer"
    inputArrivalTimer,
    --
    resultSource
    "outputArrivalTimer" "the output arrival timer"
    outputArrivalTimer,
    --
    resultSource
    "inspectionStations" "the inspection stations"
    inspectionStations,
    --
    resultSource
    "adjustmentStations" "the adjustment stations"
    adjustmentStations]
```

6.7.2 Experiment Definition

The model returns a couple of data sources. We create an experiment that shows the data from different perspectives. The names should be self-explanatory.

```
module Experiment (experiment, generators) where

import Data.Monoid

import Control.Arrow

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

import qualified Simulation.Aivika.Results.Transform as T

-- | The simulation specs.
specs = Specs { spcStartTime = 0.0,
                spcStopTime = 480.0,
                spcDT = 0.1,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

-- | The experiment.
experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1000,
    -- experimentRunCount = 10,
    experimentTitle = "Inspection and Adjustment Stations on " ++
                      "a Production Line (the Monte-Carlo simulation)" }

inputArrivalTimer  = T.ArrivalTimer $ resultByName "inputArrivalTimer"
outputArrivalTimer = T.ArrivalTimer $ resultByName "outputArrivalTimer"

inspectionStations = T.Server $ resultByName "inspectionStations"
adjustmentStations = T.Server $ resultByName "adjustmentStations"

inspectionQueue = T.Queue $ resultByName "inspectionQueue"
adjustmentQueue = T.Queue $ resultByName "adjustmentQueue"

resultProcessingTime :: ResultTransform
```

```
resultProcessingTime =
  (T.tr $ T.arrivalProcessingTime inputArrivalTimer) <>
  (T.tr $ T.arrivalProcessingTime outputArrivalTimer)

resultProcessingFactor :: ResultTransform
resultProcessingFactor =
  (T.serverProcessingFactor inspectionStations) <>
  (T.serverProcessingFactor adjustmentStations)

inspectionQueueCount      = T.queueCount inspectionQueue
inspectionQueueCountStats = T.tr $ T.queueCountStats inspectionQueue
inspectionWaitTime        = T.tr $ T.queueWaitTime inspectionQueue

adjustmentQueueCount      = T.queueCount adjustmentQueue
adjustmentQueueCountStats = T.tr $ T.queueCountStats adjustmentQueue
adjustmentWaitTime        = T.tr $ T.queueWaitTime adjustmentQueue

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView defaultInfoView,
   outputView $ defaultFinalStatsView {
     finalStatsTitle  = "Arrivals",
     finalStatsSeries = resultProcessingTime },
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "The processing factor (chart)",
     deviationChartWidth = 1000,
     deviationChartRightYSeries = resultProcessingFactor },
   outputView $ defaultFinalHistogramView {
     finalHistogramTitle = "The processing factor (histogram)",
     finalHistogramWidth = 1000,
     finalHistogramSeries = resultProcessingFactor },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "The processing factor (statistics)",
     finalStatsSeries = resultProcessingFactor },
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "The inspection queue size (chart)",
     deviationChartWidth = 1000,
     deviationChartRightYSeries =
       inspectionQueueCount <> inspectionQueueCountStats },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "The inspection queue size (statistics)",
     finalStatsSeries = inspectionQueueCountStats },
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "The inspection queue wait time (chart)",
     deviationChartWidth = 1000,
     deviationChartRightYSeries = inspectionWaitTime },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "The inspection queue wait time (statistics)",
     finalStatsSeries = inspectionWaitTime },
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "The adjustment queue size (chart)",
     deviationChartWidth = 1000,
     deviationChartRightYSeries =
       adjustmentQueueCount <> adjustmentQueueCountStats },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "The adjustment queue size (statistics)",
     finalStatsSeries = adjustmentQueueCountStats },
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "The adjustment queue wait time (chart)",
     deviationChartWidth = 1000,
     deviationChartRightYSeries = adjustmentWaitTime },
```

```
outputView $ defaultFinalStatsView {
  finalStatsTitle = "The adjustment queue wait time (statistics)",
  finalStatsSeries = adjustmentWaitTime } ]
```

6.7.3 Charting

Here you can choose one of the charting back-ends. The code is absolutely the same as it was in section 1.4.3.

6.7.4 Running Simulation Experiment

On my laptop the specified simulation experiment lasted for 15 seconds, when using the Caro-based charting backend, and it laster for 28 seconds, when using the Diagrams-based charting backed.

You can see one of the resulting charts on figure 6.1.

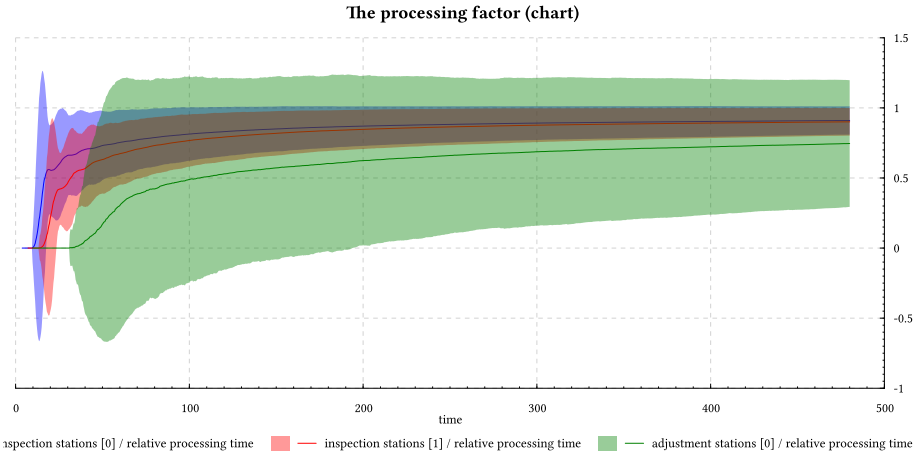


Figure 6.1: The deviation chart for the processing time.

6.8 Example: Resource Preemption

Now it is time to fulfil the promise. Below is represented the example that demonstrates the use of resource preemption. The example models the machine tools with breakdowns [11].

> Jobs arrive to a machine tool on the average of one per hour. The distribution of these interarrival times is exponential. During normal operation, the jobs are processed on a first-in, first-out basis. The time to process a job in hours is normally distributed with a mean of 0.5 and a standard deviation of 0.1. In addition to the processing time, there is a set up time that is uniformly distributed between 0.2 and 0.5 of an hour. Jobs that have been processed by the machine tool are routed to a different section of the shop and are considered to have left the machine tool area.

The machine tool experiences breakdowns during which time it can no longer process jobs. The time between breakdowns is normally distributed with a mean of 20 hours and a standard deviation of 2 hours. When a breakdown occurs, the job being processed is removed from the machine tool and is placed at the head of the queue of jobs waiting to be processed. Jobs preempted restart from the point at which they were interrupted.

When the machine tool breaks down, a repair process is initiated which is accomplished in three phases. Each phase is exponentially distributed with a mean of 3/4 of an hour. Since the repair time is the sum of independent and identically distributed exponential random variables, the repair time is Erlang distributed. The machine tool is to be analyzed for 500 hours to obtain information on the utilization of the machine tool and the time required to process a job. Statistics are to be collected for thousand simulation runs.

6.8.1 Returning Results from Model

As we did before, we write a model that returns the simulation results. Here we use the unbounded queue as well as the resource preemption considered earlier in section 3.8.

```
module Model (model) where

import Control.Monad
import Control.Monad.Trans
import Control.Category

import Data.Monoid
import Data.List

import Simulation.Aivika
import qualified Simulation.Aivika.Queue.Infinite as IQ
import qualified Simulation.Aivika.Resource.Preemption as PR

-- | How often do jobs arrive to a machine tool (exponential)?
jobArrivingMu = 1

-- | A mean of time to process a job (normal).
jobProcessingMu = 0.5

-- | The standard deviation of time to process a job (normal).
jobProcessingSigma = 0.1

-- | The minimum set-up time (uniform).
minSetUpTime = 0.2

-- | The maximum set-up time (uniform).
maxSetUpTime = 0.5

-- | A mean of time between breakdowns (normal).
breakdownMu = 20

-- | The standard deviation of time between breakdowns (normal).
breakdownSigma = 2

-- | A mean of each of the three repair phases (Erlang).
```

```
repairMu = 3/4

-- | A priority of the job (less is higher)
jobPriority = 1

-- | A priority of the breakdown (less is higher)
breakdownPriority = 0

-- | The simulation model.
model :: Simulation Results
model = do
  -- create an input queue
  inputQueue <- runEventInStartTime IQ.newFCFSQueue
  -- a counter of jobs completed
  jobsCompleted <- newArrivalTimer
  -- a counter of interrupted jobs
  jobsInterrupted <- newRef (0 :: Int)
  -- create an input stream
  let inputStream =
        randomExponentialStream jobArrivingMu
  -- create a preemptible resource
  tool <- runEventInStartTime $ PR.newResource 1
  -- the machine setting up
  machineSettingUp <-
    newPreemptibleRandomUniformServer True minSetUpTime maxSetUpTime
  -- the machine processing
  machineProcessing <-
    newPreemptibleRandomNormalServer True jobProcessingMu jobProcessingSigma
  -- the machine breakdown
  let machineBreakdown =
        do randomNormalProcess_ breakdownMu breakdownSigma
           PR.usingResourceWithPriority tool breakdownPriority $
             randomErlangProcess_ repairMu 3
           machineBreakdown
  -- start the process of breakdowns
  runProcessInStartTime machineBreakdown
  -- update a counter of job interruptions
  runEventInStartTime $
    handleSignal_ (serverTaskPreemptionBeginning machineProcessing) $ \a ->
    modifyRef jobsInterrupted (+ 1)
  -- define the queue network
  let network =
        queueProcessor
        (\a -> liftEvent $ IQ.enqueue inputQueue a)
        (IQ.dequeue inputQueue) >>>
        (withinProcessor $ PR.requestResourceWithPriority tool jobPriority) >>>
        serverProcessor machineSettingUp >>>
        serverProcessor machineProcessing >>>
        (withinProcessor $ PR.releaseResource tool) >>>
        arrivalTimerProcessor jobsCompleted
  -- start the machine tool
  runProcessInStartTime $
    sinkStream $ runProcessor network inputStream
  -- return the simulation results in start time
  return $
    results
    [resultSource
     "inputQueue" "the queue of jobs"
     inputQueue,
     --
     resultSource
     "machineSettingUp" "the machine setting up"
```

```
    machineSettingUp,
    --
    resultSource
    "machineProcessing" "the machine processing"
    machineProcessing,
    --
    resultSource
    "jobsInterrupted" "a counter of the interrupted jobs"
    jobsInterrupted,
    --
    resultSource
    "jobsCompleted" "a counter of the completed jobs"
    jobsCompleted,
    --
    resultSource
    "tool" "the machine tool"
    tool]
```

6.8.2 Experiment Definition

The simulation experiment defines 1000 runs. We are interested in the processing time, queue wait time, queue size and the server utilization, which is denoted here as the server processing factor or relative processing time.

```
module Experiment (experiment, generators) where

import Data.Monoid

import Control.Arrow

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

import qualified Simulation.Aivika.Results.Transform as T

-- | The simulation specs.
specs = Specs { spcStartTime = 0.0,
                spcStopTime = 500.0,
                spcDT = 0.1,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

-- | The experiment.
experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1000,
    -- experimentRunCount = 10,
    experimentTitle = "Machine Tool with Breakdowns" }

jobsCompleted     = T.ArrivalTimer $ resultByName "jobsCompleted"
jobsInterrupted   = resultByName "jobsInterrupted"
inputQueue        = T.Queue $ resultByName "inputQueue"
machineProcessing = T.Server $ resultByName "machineProcessing"

jobsCompletedCount =
  T.samplingStatsCount $
  T.arrivalProcessingTime jobsCompleted
```

```
processingTime :: ResultTransform
processingTime =
  T.tr $ T.arrivalProcessingTime jobsCompleted

waitTime :: ResultTransform
waitTime =
  T.tr $ T.queueWaitTime inputQueue

queueCount :: ResultTransform
queueCount =
  T.queueCount inputQueue

queueCountStats :: ResultTransform
queueCountStats =
  T.tr $ T.queueCountStats inputQueue

processingFactor :: ResultTransform
processingFactor =
  T.serverProcessingFactor machineProcessing

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView defaultInfoView,
   outputView $ defaultFinalStatsView {
     finalStatsTitle  = "Machine Tool With Breakdowns",
     finalStatsSeries = jobsCompletedCount <> jobsInterrupted },
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "The Wait Time (chart)",
     deviationChartWidth = 1000,
     deviationChartRightYSeries = waitTime },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "The Wait Time (statistics)",
     finalStatsSeries = waitTime },
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "The Queue Size (chart)",
     deviationChartWidth = 1000,
     deviationChartRightYSeries = queueCount <> queueCountStats },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "The Queue Size (statistics)",
     finalStatsSeries = queueCountStats },
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "The Processing Time (chart)",
     deviationChartWidth = 1000,
     deviationChartRightYSeries = processingTime },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "The Processing Time (statistics)",
     finalStatsSeries = processingTime },
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "The Machine Load (chart)",
     deviationChartWidth = 1000,
     deviationChartRightYSeries = processingFactor },
   outputView $ defaultFinalHistogramView {
     finalHistogramTitle = "The Machine Load (histogram)",
     finalHistogramWidth = 1000,
     finalHistogramSeries = processingFactor },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "The Machine Load (statistics)",
     finalStatsSeries = processingFactor } ]
```

6.8.3 Charting

As it became a tradition, here you can choose one of the charting back-ends. The code is absolutely the same as it was in section 1.4.3.

6.8.4 Running Simulation Experiment

When using the Cairo-based charting backend, the simulation experiment with 1000 runs lasted for 12 seconds on my laptop. But when using the Diagrams-based charting backend, the same simulation experiment lasted for 22 seconds.

On a histogram from figure 6.2, we can see the approximation of the machine load processing factor distribution.

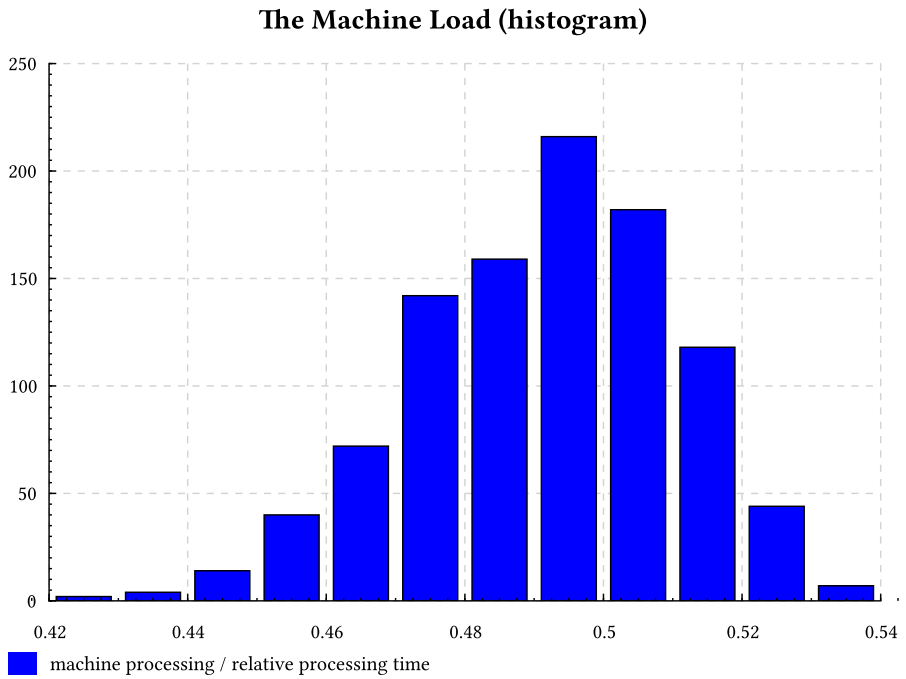


Figure 6.2: The histogram of machine load processing factor.

Chapter 7

Agent-based Modeling

Aivika provides a basic support of the agent-based modeling paradigm[15].

An idea is to try to describe a model as a cooperative behavior of a relatively large number of small agents. The agents can have states and these states can be either active or inactive. We can assign to the state a handler that is actuated under the condition that the state remains active.

7.1 Agents and Their States

We create new agents and their states within the `Simulation` computation.

```
data Agent
data AgentState

newAgent :: Simulation Agent
newState :: Agent -> Simulation AgentState
newSubstate :: AgentState -> Simulation AgentState
```

Only one of the states can be selected for each agent at the modeling time. All ancestor states remain active if they were active before, or they become active if they were deactivated. Other states are deactivated if they were active on the contrary.

```
selectedState :: Agent -> Event (Maybe AgentState)
selectState :: AgentState -> Event ()
```

The first function returns the currently selected state or `Nothing` if the agent was not yet initiated. Other function allows selecting a new state. The both functions return actions within the `Event` computation, which means that the state selection is always synchronized with the event queue.

We can assign the `Event` handlers to be performed when activating or deactivating the specified state during such a selection.

```
setStateActivation :: AgentState -> Event () -> Event ()
setStateDeactivation :: AgentState -> Event () -> Event ()
```

If the specified third state remains active when selecting another state, but the path from the old selected state to a new state goes through the third state, then we can call the third state *transitive* and can assign an action to be performed when such a transition occurs.

```
setStateTransition :: AgentState -> Event (Maybe AgentState) -> Event ()
```

Here the new selected state is sent to the corresponding `Event` computation.

What differs the agents from other simulation concepts is an ability to assign so called *timeout* and *timer handlers*. The timeout handler is an `Event` computation which is actuated in the specified time interval if the sate remains active. The timer handler is similar, but only the handler is repeated while the state still remains active. Therefore, the timeout handler accepts the time as a pure value, while the timer handler recalculates the time interval within the `Event` computation after each successful actualization.

```
addTimeout :: AgentState -> Double -> Event () -> Event ()
addTimer :: AgentState -> Event Double -> Event () -> Event ()
```

The implementation is quite simple. By the specified state handler, we create a wrapper handler which we pass in to the `enqueueEvent` function with the desired time of actuating. If the state becomes deactivated before the planned time comes then we invalidate the wrapper. After the wrapper is actuated by the event queue at the planned time, we do not call the corresponding state handler if the wrapper was invalidated earlier.

We use the `Event` computation to synchronize the agents with the event queue. It literally means that the agent-based modeling can be integrated with other simulation methods within one combined model.

7.2 Example: Agent-based Modeling

To illustrate the use of agents, let us take the Bass Diffusion model from the AnyLogic documentation [15].

> The model describes a product diffusion process. Potential adopters of a product are influenced into buying the product by advertising and by word of mouth from adopters, those who have already purchased the new product. Adoption of a new product driven by word of mouth is likewise an epidemic. Potential adopters come into contact with adopters through social interactions. A fraction of these contacts results in the purchase of the new product. The advertising causes a constant fraction of the potential adopter population to adopt each time period.

7.2.1 Returning Results From Model

Below is provided a simulation model. The agents are quite simple. They can be only in one of two possible states.

```
module Model (model) where

import Data.Array

import Control.Monad
import Control.Monad.Trans

import Simulation.Aivika
```

```
n = 100    -- the number of agents

advertisingEffectiveness = 0.011
contactRate = 100.0
adoptionFraction = 0.015

data Person = Person { personAgent :: Agent,
                       personPotentialAdopter :: AgentState,
                       personAdopter :: AgentState }

createPerson :: Simulation Person
createPerson =
  do agent <- newAgent
     potentialAdopter <- newState agent
     adopter <- newState agent
     return Person { personAgent = agent,
                     personPotentialAdopter = potentialAdopter,
                     personAdopter = adopter }

createPersons :: Simulation (Array Int Person)
createPersons =
  do list <- forM [1 .. n] $ \i ->
       do p <- createPerson
          return (i, p)
     return $ array (1, n) list

definePerson :: Person -> Array Int Person -> Ref Int -> Ref Int -> Event ()
definePerson p ps potentialAdopters adopters =
  do setStateActivation (personPotentialAdopter p) $
       do modifyRef potentialAdopters $ \a -> a + 1
          -- add a timeout
          t <- liftParameter $
               randomExponential (1 / advertisingEffectiveness)
          let st  = personPotentialAdopter p
              st' = personAdopter p
          addTimeout st t $ selectState st'
     setStateActivation (personAdopter p) $
       do modifyRef adopters  $ \a -> a + 1
          -- add a timer that works while the state is active
          let t = liftParameter $
                  randomExponential (1 / contactRate)    -- many times!
          addTimer (personAdopter p) t $
            do i <- liftParameter $
                    randomUniformInt 1 n
               let p' = ps ! i
               st <- selectedState (personAgent p')
               when (st == Just (personPotentialAdopter p')) $
                 do b <- liftParameter $
                         randomTrue adoptionFraction
                    when b $ selectState (personAdopter p')
     setStateDeactivation (personPotentialAdopter p) $
       modifyRef potentialAdopters $ \a -> a - 1
     setStateDeactivation (personAdopter p) $
       modifyRef adopters $ \a -> a - 1

definePersons :: Array Int Person -> Ref Int -> Ref Int -> Event ()
definePersons ps potentialAdopters adopters =
  forM_ (elems ps) $ \p ->
  definePerson p ps potentialAdopters adopters

activatePerson :: Person -> Event ()
```

```
activatePerson p = selectState (personPotentialAdopter p)

activatePersons :: Array Int Person -> Event ()
activatePersons ps =
  forM_ (elems ps) $ \p -> activatePerson p

model :: Simulation Results
model =
  do potentialAdopters <- newRef 0
     adopters <- newRef 0
     ps <- createPersons
     runEventInStartTime $
       do definePersons ps potentialAdopters adopters
          activatePersons ps
     return $
       results
       [resultSource
        "potentialAdopters" "potential adopters of the product" potentialAdopters,
        resultSource
        "adopters" "adopters of the product" adopters]
```

7.2.2 Experiment Definition

Unlike the previous examples, now our simulation experiment is very simple. We want to see the deviation chart for the variable number of adopters and potential adopters.

```
module Experiment (experiment, generators) where

import Data.Monoid

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 8.0,
                spcDT = 0.1,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1000,
    experimentDescription =
      "This is the famous Bass Diffusion " ++
      "model solved with help of the agent-based modelling." }

potentialAdopters = resultByName "potentialAdopters"
adopters = resultByName "adopters"

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView defaultInfoView,
   outputView $ defaultDeviationChartView {
     deviationChartLeftYSeries =
       potentialAdopters <> adopters } ]
```

7.2.3 Charting

As before, here you can choose one of the charting back-ends. The code is absolutely the same as it was in section 1.4.3.

7.2.4 Running Simulation Experiment

When using the Diagrams-based charting backed, the entire simulation experiment with 1000 runs lasted for 20 seconds on my laptop. When using the Cairo-based charting backend, it took 12 seconds only.

You can see the corresponding chart on figure 7.1.

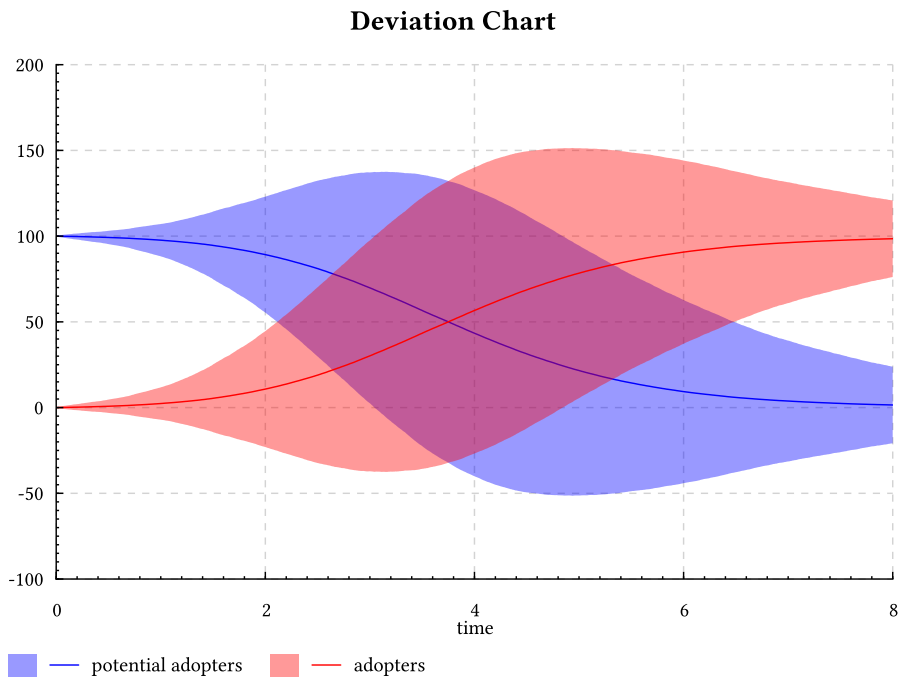


Figure 7.1: The deviation chart for variable number of adopters and potential adopters.

Chapter 8

Automata

In Aivika there are two auxiliary computations that are inspired by an idea of automata described in the literature[10, 5], which is consonant to the approach applied in Yampa[9].

8.1 Circuit

Here is the first computation that is called `Circuit`.

```
newtype Circuit a b = Circuit { runCircuit :: a -> Event (b, Circuit a b) }
```

This is an automaton that takes an input and returns the next state and output within the `Event` computation synchronized with the event queue.

The `Circuit` type is obviously an `ArrowLoop`. Therefore, we can create recursive links and introduce delays by one step using the specified initial value.

```
delayCircuit :: a -> Circuit a a
```

Also we can integrate numerically the differential equations within the circuit computation creating loopbacks using the *proc*-notation if required.

```
integCircuit :: Double  -> Circuit Double Double
```

By the specified initial value we return a circuit that treats the input as derivative and returns the integral value as output.

In a similar way we can solve numerically a system of difference equations, where the next function takes the initial value too but returns an automaton that generates the sum as output by the input specifying the difference.

```
sumCircuit :: Num a => a -> Circuit a a
```

So, the system of ODEs from section 1.3 can be rewritten as follows.

```
circuit :: Circuit () [Double]
circuit =
  let ka = 1
      kb = 1
  in proc () -> do
```

```
    rec a  <- integCircuit 100 -< - ka * a
        b  <- integCircuit 0 -< ka * a - kb * b
        c  <- integCircuit 0 -< kb * b
    returnA -< [a, b, c]
```

To get the final results of integration, now we have to transform somehow the `Circuit` arrow computation. Therefore we need some conversion.

An arbitrary circuit can be treated as the signal transform or processor.

```
circuitSignaling :: Circuit a b -> Signal a -> Signal b
circuitProcessor :: Circuit a b -> Processor a b
```

Furthermore, the circuit can be approximated in the integration time points and interpolated in other time points:

```
circuitTransform :: Circuit a b -> Transform a b
```

Here the `Transform` type is similar to the ending part of the `integ` function. It can be realized as an analogous circuit as opposite to digital one represented by the `Circuit` computation.

```
newtype Transform a b =
  Transform { runTransform :: Dynamics a -> Simulation (Dynamics b) }
```

Also it gives us another interpretation for the `Dynamics` computation. The latter can be considered as a single entity defined in all time points simultaneously, which seems to be natural as we can approximate the integral in such a way. Consider comparing this computation with the `Event` computation, where we strongly emphasize on the fact that the `Event` computation is bound up with the current simulation time point. Speaking of the `Dynamics` computation, we do not do any assumptions regarding the simulation time.

Returning to our ODE, we can run the model by approximating the circuit in the integration time points for simplicity, but not for efficiency, though.

```
model :: Simulation [Double]
model =
  do results <-
       runTransform (circuitTransform circuit) $
       return ()
     runDynamicsInStopTime results
```

This model will return almost the same results[1] but it is much slower than the model that used the `integ` function within the `Dynamics` computation.

The `integCircuit` function itself has a very small memory footprint, or more precisely, it creates a lot of small short-term functions but recycles them immediately. But this footprint is mostly neglected by the `circuitTransform` function, which is memory consuming.

On the contrary, the `integ` function may allocate a potentially huge array once, but then it consumes almost no memory when integrating numerically.

Comparing the circuit with other computations, the former always returns its output at the current modeling time without delay, but the circuit also saves its state and we can request for the next output by next input at any desired

[1] Even if we will use Euler's method which must be equivalent, there is also an inevitable inaccuracy of calculations.

time later. It is essential that the circuit allows specifying recursive links, which can be useful to describe a simple digital circuit, for example.

An approximation of the circuit in the integration time points allows using it in the differential equations. A synchronization with the event queue is provided automatically.

8.2 Network

There is a version of the `Circuit` computation, but only where the `Event` computation is replaced by the `Process` computation.

```
newtype Net a b = Net { runNet :: a -> Process (b, Net a b) }
```

The `Net` type has a more efficient implementation of the `Arrow` type class than `Processor` has. A computation received with help of the *proc*-notation must be significantly more light-weight. It is similar to the `Circuit` computation, but only `Net` is not `ArrowLoop`, because the `Process` monad is not `MonadFix`, being based on continuations.

Moreover, the `Net` computation can be easily converted to a processor and it can be done very efficiently, which shows the main use case for this type: writing some parts of the model within `Net` computation using the *proc*-notation with the further conversion.

```
netProcessor :: Net a b -> Processor a b
netProcessor = Processor . loop
  where loop x as =
          Cons $
          do (a, as') <- runStream as
             (b, x') <- runNet x a
             return (b, loop x' as')
```

The problem is that the `Net` type has no clear multiplexing and demultiplexing facilities for parallelizing the processing. Also its opposite conversion is quite costly and there is actually no guarantee that the specified processor will produce exactly one output for each input value.

```
processorNet :: Processor a b -> Net a b
```

Nevertheless, the both computations can be useful in their combination.

Chapter 9

System Dynamics

Earlier we introduced the `integ` function that allows approximating integrals. There is the similar function `diffsum` that allows defining difference equations. They are a foundation of System Dynamics. In this chapter we will consider some examples related to this field of simulation.

9.1 Example: Parametric Model

Now we will focus on a practical question: how to prepare a parametric model for the Monte-Carlo simulation? For example, it can be useful for providing the sensitivity analysis.

Let us take the financial model[17] described in Vensim 5 Modeling Guide, Chapter Financial Modeling and Risk. Probably, the best way to describe the model is just to show its equations.

The equations use the `npv` function from System Dynamics. It returns the Net Present Value (NPV) of the stream computed using the specified discount rate, the initial value and some factor (usually 1).

```
npv :: Dynamics Double                  -- ^ the stream
       -> Dynamics Double               -- ^ the discount rate
       -> Dynamics Double               -- ^ the initial value
       -> Dynamics Double               -- ^ factor
       -> Simulation (Dynamics Double)  -- ^ the Net Present Value (NPV)
npv stream rate init factor =
  mdo let dt' = liftParameter dt
      df <- integ (- df * rate) 1
      accum <- integ (stream * df) init
      return $ (accum + dt' * stream * df) * factor
```

Also we need a helper conditional combinator that allows avoiding the *do*-notation in some cases.

```
-- | Implement the if-then-else operator.
ifDynamics :: Dynamics Bool -> Dynamics a -> Dynamics a -> Dynamics a
ifDynamics cond x y =
  do a <- cond
     if a then x else y
```

9.1.1 Returning Results From Model

After we finished the necessary preliminaries, now we can show how the parametric model can be prepared for the Monte-Carlo simulation.

We represent each external parameter as a `Parameter` computation. To be reproducible within every simulation run, the random parameter must be memoized with help of the `memoParameter` function.

Also this model returns the `Results` within the `Simulation` computation. Such results can be processed then or just printed in Terminal.

```
{-# LANGUAGE RecursiveDo #-}

module Model
       (-- * Simulation Model
        model,
        -- * Variable Names
        netIncomeName,
        netCashFlowName,
        npvIncomeName,
        npvCashFlowName,
        -- * External Parameters
        Parameters(..),
        defaultParams,
        randomParams) where

import Control.Monad

import Simulation.Aivika
import Simulation.Aivika.SystemDynamics
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

-- | The model parameters.
data Parameters =
  Parameters { paramsTaxDepreciationTime    :: Parameter Double,
               paramsTaxRate                :: Parameter Double,
               paramsAveragePayableDelay    :: Parameter Double,
               paramsBillingProcessingTime  :: Parameter Double,
               paramsBuildingTime           :: Parameter Double,
               paramsDebtFinancingFraction  :: Parameter Double,
               paramsDebtRetirementTime     :: Parameter Double,
               paramsDiscountRate           :: Parameter Double,
               paramsFractionalLossRate     :: Parameter Double,
               paramsInterestRate           :: Parameter Double,
               paramsPrice                  :: Parameter Double,
               paramsProductionCapacity     :: Parameter Double,
               paramsRequiredInvestment     :: Parameter Double,
               paramsVariableProductionCost :: Parameter Double }

-- | The default model parameters.
defaultParams :: Parameters
defaultParams =
  Parameters { paramsTaxDepreciationTime    = 10,
               paramsTaxRate                = 0.4,
               paramsAveragePayableDelay    = 0.09,
               paramsBillingProcessingTime  = 0.04,
               paramsBuildingTime           = 1,
               paramsDebtFinancingFraction  = 0.6,
               paramsDebtRetirementTime     = 3,
               paramsDiscountRate           = 0.12,
               paramsFractionalLossRate     = 0.06,
```

```
              paramsInterestRate           = 0.12,
              paramsPrice                  = 1,
              paramsProductionCapacity     = 2400,
              paramsRequiredInvestment     = 2000,
              paramsVariableProductionCost = 0.6 }

-- | Random parameters for the Monte-Carlo simulation.
randomParams :: IO Parameters
randomParams =
  do averagePayableDelay    <- memoParameter $ randomUniform 0.07 0.11
     billingProcessingTime  <- memoParameter $ randomUniform 0.03 0.05
     buildingTime           <- memoParameter $ randomUniform 0.8 1.2
     fractionalLossRate     <- memoParameter $ randomUniform 0.05 0.08
     interestRate           <- memoParameter $ randomUniform 0.09 0.15
     price                  <- memoParameter $ randomUniform 0.9 1.2
     productionCapacity     <- memoParameter $ randomUniform 2200 2600
     requiredInvestment     <- memoParameter $ randomUniform 1800 2200
     variableProductionCost <- memoParameter $ randomUniform 0.5 0.7
     return defaultParams { paramsAveragePayableDelay    = averagePayableDelay,
                            paramsBillingProcessingTime  = billingProcessingTime,
                            paramsBuildingTime           = buildingTime,
                            paramsFractionalLossRate     = fractionalLossRate,
                            paramsInterestRate           = interestRate,
                            paramsPrice                  = price,
                            paramsProductionCapacity     = productionCapacity,
                            paramsRequiredInvestment     = requiredInvestment,
                            paramsVariableProductionCost =
                              variableProductionCost }

-- | This is the model itself that returns experimental data.
model :: Parameters -> Simulation Results
model params =
  mdo let getParameter f = liftParameter $ f params

      -- the equations below are given in an arbitrary order!

      bookValue <- integ (newInvestment - taxDepreciation) 0
      let taxDepreciation = bookValue / taxDepreciationTime
          taxableIncome = grossIncome - directCosts - losses
                          - interestPayments - taxDepreciation
          production = availableCapacity
          availableCapacity = ifDynamics (time .>=. buildingTime)
                              productionCapacity 0
          taxDepreciationTime = getParameter paramsTaxDepreciationTime
          taxRate = getParameter paramsTaxRate
      accountsReceivable <- integ (billings - cashReceipts - losses)
                            (billings / (1 / averagePayableDelay
                                         + fractionalLossRate))
      let averagePayableDelay = getParameter paramsAveragePayableDelay
      awaitingBilling <- integ (price * production - billings)
                         (price * production * billingProcessingTime)
      let billingProcessingTime = getParameter paramsBillingProcessingTime
          billings = awaitingBilling / billingProcessingTime
          borrowing = newInvestment * debtFinancingFraction
          buildingTime = getParameter paramsBuildingTime
          cashReceipts = accountsReceivable / averagePayableDelay
      debt <- integ (borrowing - principalRepayment) 0
      let debtFinancingFraction = getParameter paramsDebtFinancingFraction
          debtRetirementTime = getParameter paramsDebtRetirementTime
          directCosts = production * variableProductionCost
          discountRate = getParameter paramsDiscountRate
          fractionalLossRate = getParameter paramsFractionalLossRate
```

```
          grossIncome = billings
          interestPayments = debt * interestRate
          interestRate = getParameter paramsInterestRate
          losses = accountsReceivable * fractionalLossRate
          netCashFlow = cashReceipts + borrowing - newInvestment
                        - directCosts - interestPayments
                        - principalRepayment - taxes
          netIncome = taxableIncome - taxes
          newInvestment = ifDynamics (time .>=. buildingTime)
                          0 (requiredInvestment / buildingTime)
      npvCashFlow <- npv netCashFlow discountRate 0 1
      npvIncome <- npv netIncome discountRate 0 1
      let price = getParameter paramsPrice
          principalRepayment = debt / debtRetirementTime
          productionCapacity = getParameter paramsProductionCapacity
          requiredInvestment = getParameter paramsRequiredInvestment
          taxes = taxableIncome * taxRate
          variableProductionCost = getParameter paramsVariableProductionCost

      return $
        results
        [resultSource netIncomeName "Net income" netIncome,
         resultSource netCashFlowName "Net cash flow" netCashFlow,
         resultSource npvIncomeName "NPV income" npvIncome,
         resultSource npvCashFlowName "NPV cash flow" npvCashFlow]

-- the names of the variables we are interested in
netIncomeName   = "netIncome"
netCashFlowName = "netCashFlow"
npvIncomeName   = "npvIncome"
npvCashFlowName = "npvCashFlow"
```

Now we can apply the Monte-Carlo simulation to this parametric model, for example, to define how sensitive are some variables to the random external parameters.

The point is that not only ODEs can be parametric. There is not any difference, whether we integrate numerically, or run the discrete event simulation, or simulate the agents. The external parameters are just `Parameter` computations that can be used within other simulation computations.

9.1.2 Experiment Definition

Unlike other examples, here we actually define two experiments: one for the Monte-Carlo simulation and another more simple with single run.

```
module Experiment (monteCarloExperiment, singleExperiment,
                   monteCarloGenerators, singleGenerators) where

import Control.Monad

import Data.Monoid

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

import Model

-- the simulation specs
```

```
specs = Specs 0 5 0.015625 RungeKutta4 SimpleGenerator

-- | The experiment for the Monte-Carlo simulation.
monteCarloExperiment :: Experiment
monteCarloExperiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1000,
    experimentTitle = "Financial Model (the Monte-Carlo simulation)",
    experimentDescription =
      "Financial Model (the Monte-Carlo simulation) as described in " ++
      "Vensim 5 Modeling Guide, Chapter Financial Modeling and Risk." }

netIncome = resultByName netIncomeName
npvIncome = resultByName npvIncomeName

netCashFlow = resultByName netCashFlowName
npvCashFlow = resultByName npvCashFlowName

monteCarloGenerators :: ChartRendering r => [WebPageGenerator r]
monteCarloGenerators =
  [outputView defaultExperimentSpecsView,
   outputView defaultInfoView,
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "Chart 1",
     deviationChartPlotTitle =
       "The deviation chart for Net Income and Cash Flow",
     deviationChartLeftYSeries = netIncome <> netCashFlow },
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "Chart 2",
     deviationChartPlotTitle =
       "The deviation chart for Net Present Value of Income and Cash Flow",
     deviationChartLeftYSeries = npvIncome <> npvCashFlow },
   outputView $ defaultFinalHistogramView {
     finalHistogramTitle = "Histogram 1",
     finalHistogramPlotTitle ="Histogram for Net Income and Cash Flow",
     finalHistogramSeries = netIncome <> netCashFlow },
   outputView $ defaultFinalHistogramView {
     finalHistogramTitle = "Histogram 2",
     finalHistogramPlotTitle =
       "Histogram for Net Present Value of Income and Cash Flow",
     finalHistogramSeries = npvIncome <> npvCashFlow },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "Summary 1",
     finalStatsSeries = netIncome <> netCashFlow },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "Summary 2",
     finalStatsSeries = npvIncome <> npvCashFlow } ]

-- | The experiment with single simulation run.
singleExperiment :: Experiment
singleExperiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentTitle = "Financial Model",
    experimentDescription =
      "Financial Model as described in " ++
      "Vensim 5 Modeling Guide, Chapter Financial Modeling and Risk." }

singleGenerators :: ChartRendering r => [WebPageGenerator r]
singleGenerators =
  [outputView defaultExperimentSpecsView,
```

```
  outputView defaultInfoView,
  outputView $ defaultTimeSeriesView {
    timeSeriesTitle = "Time Series 1",
    timeSeriesPlotTitle = "Time series of Net Income and Cash Flow",
    timeSeriesLeftYSeries = netIncome <> netCashFlow },
  outputView $ defaultTimeSeriesView {
    timeSeriesTitle = "Time Series 2",
    timeSeriesPlotTitle =
      "Time series of Net Present Value for Income and Cash Flow",
    timeSeriesLeftYSeries = npvIncome <> npvCashFlow },
  outputView $ defaultTableView {
    tableTitle = "Table",
    tableSeries = netIncome <> netCashFlow <> npvIncome <> npvCashFlow } ]
```

9.1.3 Charting

Similarly, we need to have different charting applications to run both our simulation experiments one by one.

Cairo-based Charting Back-end

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Cairo

import Graphics.Rendering.Chart.Backend.Cairo

import Model
import Experiment

main = do

  -- run the ordinary simulation
  putStrLn "*** The simulation with default parameters..."
  runExperiment
    singleExperiment singleGenerators
    (WebPageRenderer (CairoRenderer PNG) experimentFilePath) (model defaultParams)
  putStrLn ""

  -- run the Monte-Carlo simulation
  putStrLn "*** The Monte-Carlo simulation..."
  randomParams >>= runExperimentParallel
    monteCarloExperiment monteCarloGenerators
    (WebPageRenderer (CairoRenderer PNG) experimentFilePath) . model
```

Diagrams-based Charting Back-end

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Diagrams

import Graphics.Rendering.Chart.Backend.Diagrams


import Model
import Experiment

main = do
  fonts <- loadCommonFonts
  let renderer = DiagramsRenderer SVG (return fonts)
```

```
-- run the ordinary simulation
putStrLn "*** The simulation with default parameters..."
runExperiment
  singleExperiment singleGenerators
  (WebPageRenderer renderer experimentFilePath) (model defaultParams)
putStrLn ""

-- run the Monte-Carlo simulation
putStrLn "*** The Monte-Carlo simulation..."
randomParams >>= runExperimentParallel
  monteCarloExperiment monteCarloGenerators
  (WebPageRenderer renderer experimentFilePath) . model
```

9.1.4 Running Simulation Experiments

When using the Diagrams-based charting backend, the both simulation experiments one by one, where the former contained a single run but the latter consisted of 1000 runs, lasted for 34 seconds on my laptop. When using the Cairo-based charting backend, the both experiments lasted for 23 seconds.

You can see one of the charts on figure 9.1.

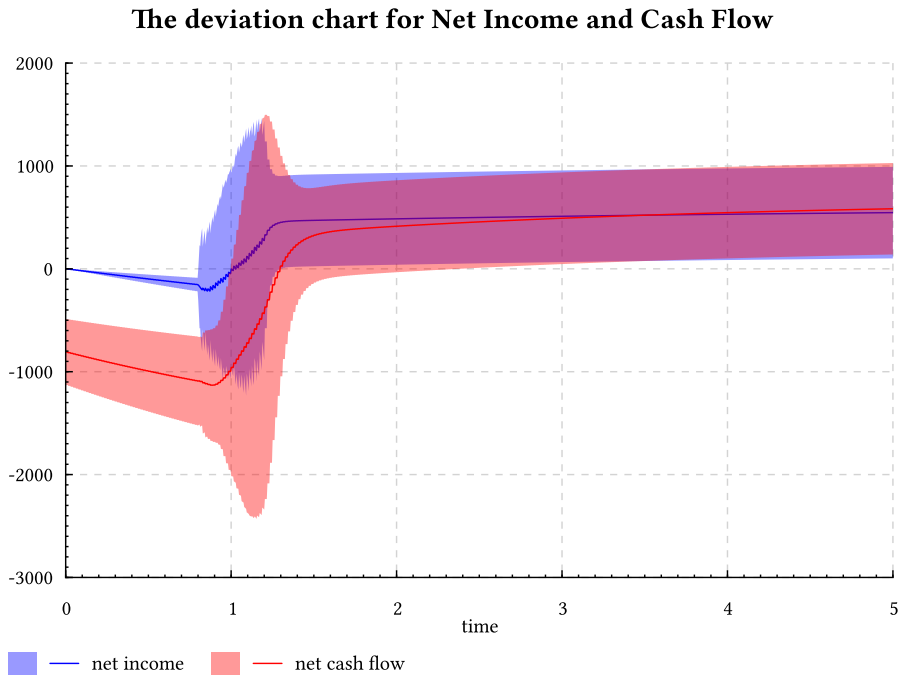


Figure 9.1: The deviation chart for Net income and Cache Flow.

9.2 Example: Using Arrays

Some vendors offer different versions of their simulation software tools, where one of the main advantages of using a commercial version is an ability to use

arrays.

There is no need in special support of arrays in Aivika. They can be naturally used with the simulation computations.

9.2.1 Returning Results from Model

Let us take model Linear Array from Berkeley Madonna[7] to demonstrate the main idea.

```
{-# LANGUAGE RecursiveDo #-}

module Model (model) where

import Data.Array
import Control.Monad
import Control.Monad.Trans

import qualified Data.Vector as V

import Simulation.Aivika
import Simulation.Aivika.SystemDynamics
import Simulation.Aivika.Experiment

-- | This is an analog of 'V.generateM' included in the Haskell platform.
generateArray :: (Ix i, Monad m) => (i, i) -> (i -> m a) -> m (Array i a)
generateArray bnds generator =
  do ps <- forM (range bnds) $ \i ->
       do x <- generator i
          return (i, x)
     return $ array bnds ps

model :: Int -> Simulation Results
model n =
  mdo m <- generateArray (1, n) $ \i ->
        integ (q + k * (c!(i - 1) - c!i) + k * (c!(i + 1) - c!i)) 0
      let c =
            array (0, n + 1) [(i, if (i == 0) || (i == n + 1)
                                  then 0
                                  else (m!i / v)) | i <- [0 .. n + 1]]
          q = 1
          k = 2
          v = 0.75
      return $ results
        [resultSource "t" "time" time,
         resultSource "m" "M" m,
         resultSource "c" "C" c]
```

The code provided above uses the standard `Array` module. If we used the `Vector` module, then we would need no function `generateArray` at all.

This model creates an array of integrals. Similarly, we could use arrays in the discrete event simulation or agent-based model.

9.2.2 Experiment Definition

In the experiment we want to look at the arrays from different perspectives.

```
module Experiment (experiment, generators) where

import Data.Monoid
```

```
import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

specs = Specs { spcStartTime = 0,
                spcStopTime = 500,
                spcDT = 0.1,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1,
    experimentTitle = "Linear Array",
    experimentDescription = "Model Linear Array as described in " ++
                            "the examples included in Berkeley-Madonna." }

t = resultByName "t"
m = resultByName "m"
c = resultByName "c"

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView $ defaultTableView {
     tableSeries = t <> m <> c },
   outputView $ defaultTimeSeriesView {
     timeSeriesLeftYSeries = m,
     timeSeriesWidth = 800,
     timeSeriesHeight = 800 },
   outputView $ defaultTimeSeriesView {
     timeSeriesRightYSeries = c,
     timeSeriesWidth = 800,
     timeSeriesHeight = 800 },
   outputView $ defaultTimeSeriesView {
     timeSeriesLeftYSeries = m,
     timeSeriesRightYSeries = c,
     timeSeriesWidth = 800,
     timeSeriesHeight = 800 },
   outputView $ defaultXYChartView {
     xyChartXSeries = t,
     xyChartLeftYSeries = m,
     xyChartWidth = 800,
     xyChartHeight = 800 },
   outputView $ defaultXYChartView {
     xyChartXSeries = t,
     xyChartRightYSeries = c,
     xyChartWidth = 800,
     xyChartHeight = 800 },
   outputView $ defaultXYChartView {
     xyChartXSeries = t,
     xyChartLeftYSeries = m,
     xyChartRightYSeries = c,
     xyChartWidth = 800,
     xyChartHeight = 800 } ]
```

9.2.3 Charting

Since our model depends on the numerical parameter, we define other charting applications, although they are very similar to those ones that we used before.

Cairo-based Charting Back-end

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Cairo

import Graphics.Rendering.Chart.Backend.Cairo

import Model
import Experiment

main =
  runExperiment experiment generators
  (WebPageRenderer (CairoRenderer PNG) experimentFilePath)
  (model 51)
```

Diagrams-based Charting Back-end

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Diagrams

import Graphics.Rendering.Chart.Backend.Diagrams

import Model
import Experiment

main =
  do fonts <- loadCommonFonts
     let renderer = DiagramsRenderer SVG (return fonts)
     runExperiment experiment generators
       (WebPageRenderer renderer experimentFilePath)
       (model 51)
```

Also there is no need to run the simulation experiment in parallel as there is only 1 simulation run.

9.2.4 Running Simulation Experiment

When using the Diagrams-based charting backend, the single simulation run lasted for 3 minutes 57 seconds, while it lasted only for 14 seconds, when using the Cairo-based charting backend. Such a huge difference is related to the fact the SVG vector graphics files contain too much information. Actually, there was no need to use such a small integration time step. Please be careful about this pitfall!

For example, after increasing the integration time step in 2 times, the simulation experiment lasted for 1 minute 25 seconds, when using the Diagrams-based charting backend.

You can see one of the corresponding charts on figure 9.2 with the original time step.

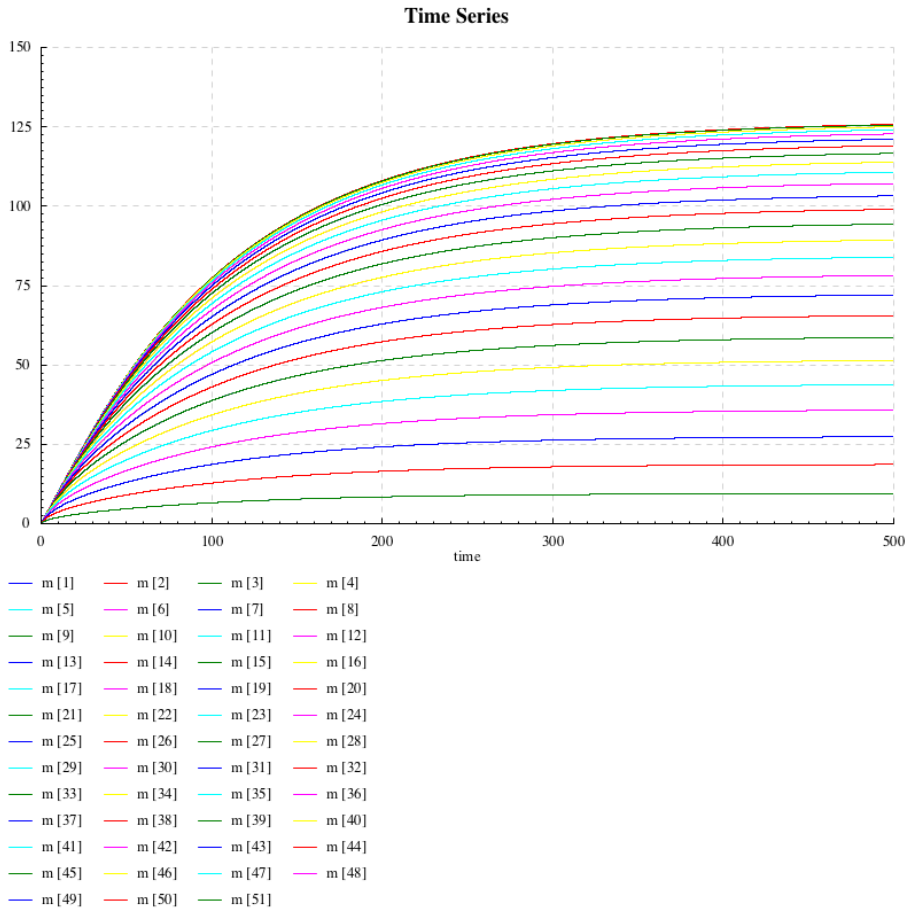


Figure 9.2: The time series for arrays.

Chapter 10

GPSS-like DSL

Aivika supports an internal domain-specific language (DSL), which is similar to the popular GPSS modeling language[13], but this is the same Haskell.

Note that the GPSS-like DSL is not equivalent to the original GPSS language, but it may return very similar results in some cases, while it can also return quite different results in other cases.

The `aivika-gpss` package implements the most of GPSS simulation blocks, but the main difference is as follows.

Like GPSS, the package tries to treat the transact priorities properly within each block. Here it works in a very similar way even for such non-trivial blocks as `PREEMPT`, `GATHER` and `ASSEMBLE`. But unlike GPSS, the blocks behave independently from each other, where the transact priorities are not used when deciding which of the blocks will be activated next. The order of activating the blocks is unpredictable.

10.1 Blocks and Transacts

The *blocks* and *generators* are basic computations of the GPSS-like DSL. They have the following definition[1].

```
data Block a b = Block { blockProcess :: a -> Process b }

newtype GeneratorBlock a =
  GeneratorBlock { runGeneratorBlock :: Block a () -> Process () }
```

This is the next elaboration of an idea of using the `Process` computation as a composing and building unit. As you might remember, `Process` denotes the disontinuous process.

Here, the `Block` computation corresponds to the GPSS block that has input of type `a` and output of type `b`. The `GeneratorBlock` computation corresponds to the `GENERATE` construct from GPSS that, being applied to some terminating `Block`, returns an action that models the traversing of the generated items through the corresponded block. The block is *terminating* if its output, i.e. the second type parameter, is unit `()` like this: `Block a ()`.

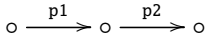
[1] A reader experienced in Haskell might say that the `Block` computation is actually a Kleisli arrow, but without `ArrowLoop`, because `Process` is not `MonadFix`.

So, the `TERMINATE` construct from GPSS will correspond to the following computation:

```
terminateBlock :: Block a ()
```

Since `Block` is a `Category`, we can connect blocks in the chain:

```
p1 :: Block a b
p2 :: Block b c

p3 :: Block a c
p3 = p1 >>> p2
```



To create generators, we can use the `Stream` computation, especially, random streams.

```
streamGeneratorBlock :: Stream (Arrival a)
                        -> Int
                        -> GeneratorBlock (Transact a)

streamGeneratorBlock0 :: Stream (Arrival a)
                         -> GeneratorBlock (Transact a)
```

The first function accepts the input stream and an integer priority that will be assigned to new transacts. The second function assigns the zero priority. The both functions return a generator of transacts.

Each transact has an integer priority and bears some value.

```
data Transact a

transactPriority :: Transact a -> Int
transactValue :: Transact a -> a
```

It is very important to note that the `Block` computation can process the transacts in parallel unlike the `Stream` computation.

The most of simulation GPSS blocks have direct or very similar analogs in Aivika. In other cases, they can be manually represented as `Block` computations, for example, in case of the `SELECT` block that requires the information about queues and other entities it connects to.

Table 10.1: The GPSS block correspondence.

GPSS Construct	An analog in Aivika
`GENERATE`	`streamGeneratorBlock`
`TERMINATE`	`terminateBlock`
a chain of blocks	a chain of computations
`ADVANCE`	`advanceBlock`
`ASSEMBLE`	`assembleBlock`
`ASSIGN`	`assignBlock`
`DEPART`	`departBlock`
`ENTER`	`enterBlock`

GATHER	gatherBlock
LEAVE	leaveBlock
LINK	linkBlock
LOOP	loopBlock
MATCH	matchBlock
PREEMPT	preemptBlock
PRIORITY	priorityBlock
QUEUE	queueBlock
RELEASE	releaseBlock
RETURN	returnBlock
SEIZE	seizeBlock
SPLIT	splitBlock
TEST	awaitingTestBlock / transferringTestBlock
UNLINK	unlinkBlock

The point is that we can naturally combine other computations with `Block` computations. For example, it literally means that we can use agents within the GPSS simulation model and vice versa.

10.2 Example: Using GPSS

Below is considered a model[13] that demonstrates the use of the GPSS-like DSL.

> The professor consults students one by one in auditorium. The consultation can be interrupted by telephone call. The time unit is 0.01 minutes. The students come in the FIFO order. The telephone calls every 20 +/- 5 minutes. It lasts for a time interval distributed exponentially with average value 2 minutes. The consultation time with one student has exponential distribution with average value 10 minutes. When interrupting, the consultation time increases by 3 minutes to restore the context of the interrupted talk.

The GPSS model is as follows. For simplicity, we assume that there are 20 students awaiting the consultation.

```
        GENERATE 2000,500,,,1
        GATE NI PROF,Busy
        PREEMPT PROF,PR,Add,5
        ADVANCE (Exponential(1,0,200))
        RETURN PROF
Busy    TERMINATE

        GENERATE ,,,20
        QUEUE LINE
        SEIZE PROF
        DEPART LINE
        ADVANCE (Exponential(1,0,1000))
LetGo   RELEASE PROF
        TERMINATE
```

```
Add     ASSIGN 5+,300
        ADVANCE P5
        TRANSFER ,LetGo

        GENERATE 20000
        TERMINATE 1

        START 1
```

Here we see the preemption block that models the telephone call. It essentially complicates the model. Fortunately, there is a direct analog in Aivika.

10.2.1 Returning Results from Model

Our model is very similar, but only it is written in pure Haskell. Also we reset the statistics at time 4000.

```
module Model (model) where

import Prelude hiding (id)

import Control.Category
import Control.Monad.Trans

import Data.Maybe

import Simulation.Aivika
import Simulation.Aivika.GPSS
import qualified Simulation.Aivika.GPSS.Queue as Q

model :: Simulation Results
model =
  do line <- runEventInStartTime Q.newQueue
     prof <- runEventInStartTime newFacility

     let phoneCallStream = randomUniformStream (2000 - 500) (2000 + 500)
         studentStream   = takeStream 20 $ randomUniformStream 0 0

     let phoneCalls     = streamGeneratorBlock phoneCallStream 1
         phoneCallChain =
           Block (\a ->
                   do f <- liftEvent (facilityInterrupted prof)
                      if f
                        then blockProcess (transferBlock busy) a
                        else return a) >>>
           preemptBlock prof
           (PreemptBlockMode { preemptBlockPriorityMode = True,
                               preemptBlockTransfer     = Just add,
                               -- preemptBlockTransfer     = Nothing,
                               preemptBlockRemoveMode   = False }) >>>
           advanceBlock (randomExponentialProcess_ 200) >>>
           returnBlock prof >>>
           busy
         busy           = terminateBlock

         students       = streamGeneratorBlock studentStream 0
         studentChain   =
           queueBlock line 1 >>>
           seizeBlock prof >>>
           departBlock line 1 >>>
           advanceBlock (randomExponentialProcess_ 1000) >>>
```

```
          letGo
        letGo          =
          releaseBlock prof >>>
          terminateBlock
        add dt0        =
          let dt = maybe 0 id dt0
          in advanceBlock (holdProcess (dt + 300)) >>>
             transferBlock letGo

    runProcessInStartTime $
      runGeneratorBlock phoneCalls phoneCallChain

    runProcessInStartTime $
      runGeneratorBlock students studentChain

    runEventInStartTime $
      enqueueEvent 4000 $
      do Q.resetQueue line
         resetFacility prof

    return $
      results
      [resultSource "line" "Line" line,
       resultSource "prof" "Prof" prof]
```

10.2.2 Experiment Definition

In the experiment we want to see some statistics, for example, how the queue size will decrease. The corresponding series is called `lineContent`.

```
{-# LANGUAGE FlexibleContexts #-}

module Experiment (experiment, generators) where

import Data.Monoid

import Control.Arrow

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

import qualified Simulation.Aivika.Results.Transform as T
import qualified Simulation.Aivika.GPSS.Results.Transform as GpssT

-- | The simulation specs.
specs = Specs { spcStartTime = 0.0,
                spcStopTime = 20000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

-- | The experiment.
experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 10000,
    -- experimentRunCount = 100,
    experimentTitle = "The GPSS module 7.31" }
```

```
line = GpssT.Queue $ resultByName "line"
lineContent = GpssT.queueContent line
lineContentStats = T.tr $ GpssT.queueContentStats line
lineWaitTime = T.tr $ GpssT.queueWaitTime line
lineNonZeroEntryWaitTime = T.tr $ GpssT.queueNonZeroEntryWaitTime line

prof = GpssT.Facility $ resultByName "prof"
profUtilCount = GpssT.facilityUtilisationCount prof
profUtilCountStats = T.tr $ GpssT.facilityUtilisationCountStats prof
profHoldingTime = T.tr $ GpssT.facilityHoldingTime prof

statsView title series =
  defaultFinalStatsView {
    finalStatsTitle = title,
    finalStatsSeries = series
  }

chartView title series =
  defaultDeviationChartView {
    deviationChartTitle = title,
    deviationChartRightYSeries = series
  }

histogramView title series =
  defaultFinalHistogramView {
    finalHistogramTitle = title,
    finalHistogramSeries = series
  }

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView defaultInfoView,
   outputView $ statsView "PROF Utilisation" profUtilCount,
   outputView $ chartView "PROF Utilisation" profUtilCount,
   outputView $ statsView "PROF Holding Time" profHoldingTime,
   outputView $ chartView "PROF Holding Time" profHoldingTime,
   outputView $ statsView "LINE Content" lineContent,
   outputView $ chartView "LINE Content" lineContent,
   outputView $ histogramView "LINE Content" lineContent,
   outputView $ statsView "LINE Wait Time" $
     lineWaitTime <>
     lineNonZeroEntryWaitTime,
   outputView $ chartView "LINE Wait Time" $
     lineWaitTime <>
     lineNonZeroEntryWaitTime]
```

10.2.3 Charting

To run the simulation, you can choose one of the charting back-ends. The code is absolutely the same as it was in section 1.4.3.

10.2.4 Running Simulation Experiment

When using the Diagrams-based charting backed, the whole simulation experiment with 10000 (ten thousands) runs lasted for 2 minutes 16 seconds on my MacBook Pro. When using the Cairo-based charting backend, the same simulation experiment lasted for 1 minute 56 seconds.

You can see the deviation chart for a number of awaiting students on figure 10.1.

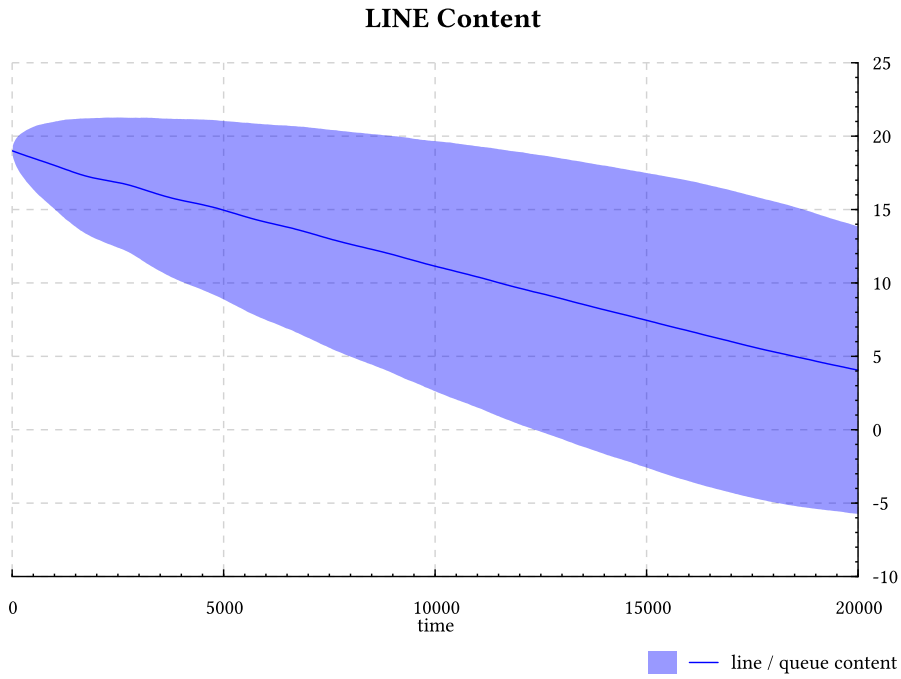


Figure 10.1: The number of students awaiting the consultation.

Part II

Parallel and Distributed Simulation

In this part the parallel and distributed simulation based on the optimistic Time Warp method is considered. It can be useful if you are going to utilize the modern multi-core processor computers or multi-computer clusters. It is even possible to connect computers located remotely and separated by far distance for creating a distributed simulation, for example, where the cluster nodes can be located in different remote computer centres connected through the Internet.

For that, Aivika supports a mode that allows recovering the distributed simulation after temporary connection errors that are inevitable in complex environments, but it works within the specified time-out limits after exceeding which the broken distributed simulation stops automatically. It allows creating safe and robust computational services based on the Linux operating system with the desired response characteristics.

Moreover, there are tasks that cannot be allocated in memory of single computer. These tasks can be solved only by creating the distributed simulation model.

Chapter 11

Generalizing Simulation

All simulation computations considered before can be generalized to be useful in other types of simulation, in particular, in distributed simulation. This approach is so general that it works perfectly for nested simulation too.

11.1 Two Versions of Simulation Library

Actually, there are two versions of the Aivika simulation library. There is a basic version optimized for sequential simulation. This is the `aivika` package in the distribution. There is also a generalized version, which can be adapted to other types of simulation. This is already the `aivika-transformers` package.

The `aivika` package could be a particular case of `aivika-transformers`, but it was stayed alone for different reasons. At first, `aivika` has a more simple documentation, which is very important. Also it is faster a little bit, although the difference in speed can decrease in the future. Also `aivika` covers the most of use cases that the modeler can confront with in practice. So, it was decided to keep the `aivika` package, although it could be fully replaced by `aivika-transformers`.

Earlier in the book we used the `aivika` package, but the text below is mainly related to the `aivika-transformers` package.

11.2 Replacing IO with Abstract Computation

If we'll repeat the `Event` computation definition from the first part devoted to sequential simulation, then we'll see that it returns a value in the `IO` monad.

```
newtype Event a = Event (Point -> IO a)
```

The generalized version contains another computation with the same name `Event` but parameterized by some other computation that corresponds to a variable type `m`.

```
newtype Event m a = Event (Point -> m a)
```

This generalized computation along with others are often either monad transformers or very similar to them, because of which the corresponding `aivika-transformers` package has such a name.

To distinguish different computations with the same names, the generalized version resides in another module namespace:

```
module Simulation.Aivika.Trans
```

This is not very idiomatic approach for Haskell, but it works well as it makes easy to convert the sequential models to distributed and nested ones, which can be important, if we are going to develop a distributed model step by step, starting from the sequential prototype model and then extending it to the distributed one.

So, not only the computations retain their original names, but also the corresponding functions have almost the same signatures, generalized only and with constraints.

```
enqueueEvent :: EventQueueing m => Double -> Event m () -> Event m ()
```

The `EventQueueing` type class literally means that its instance implements the event queue. There are also different `Ref` type classes, where their instances implement a mutable reference with the corresponding guarantees. There is a `MonadException` type class too, where its instance must implement the `IO` exception handling.

These three constraints are essentially encompassed by the `MonadDES` type class: (1) an ability to enqueue events, (2) an ability to create and mutate references and (3) an ability to handle `IO` exceptions. Also (4) the instance must be a monad.

```
class MonadDES m
```

Further, we will assume that our simulation computations such as `Event`, `Process` and others are parameterized by some `MonadDES` computation.

There is also a similar `MonadSD` type class that allows integrating differential equations, but it is rarely used.

```
class MonadSD m
```

It is important that the standard `IO` imperative monad is an instance of the both type classes, which actually means that the `aivika` package could be replaced by its generalized sibling `aivika-transformers`.

```
instance MonadDES IO
instance MonadSD IO
```

Only when using `IO` in the generalized simulation computations, we have also to import a module that contains the corresponding instances:

```
import Simulation.Aivika.IO
```

There are other corresponding modules for distributed simulation computation and nested simulation computation.

11.3 Generalizing Sequential Model

Provided the sequential simulation model, we can generalize it to be suitable for other types of simulation such as distributed simulation or nested one.

We usually have to import another Aivika library module namespace by replacing `Simulation.Aivika` with `Simulation.Aivika.Trans`. Also we have to add the parameter computation to simulation computations.

For example, we can introduce a type synonym like this:

```
type DES = IO
```

Then we can use type `Event DES a` instead of `Event a` in the model.

Let's take our sequential model from section 2.6. It is definition is repeated below.

```
module Model(model) where

import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
meanRepairTime = 0.5

model :: Simulation Results
model =
  do totalUpTime <- newRef 0.0

     let machine :: Process ()
         machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)
              repairTime <-
                randomExponentialProcess meanRepairTime
              machine

     runProcessInStartTime machine
     runProcessInStartTime machine

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

     return $
       results
       [resultSource
        "upTimeProp"
        "The long-run proportion of up time (~ 0.66)"
        upTimeProp]
```

Now by applying the procedure described in this section, we can generalize this sequential simulation model.

```
module Model(model) where

import Control.Monad.Trans
```

```
import Simulation.Aivika.Trans
import Simulation.Aivika.IO

type DES = IO

meanUpTime = 1.0
meanRepairTime = 0.5

model :: Simulation DES (Results DES)
model =
  do totalUpTime <- newRef 0.0

     let machine :: Process DES ()
         machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)
              repairTime <-
                randomExponentialProcess meanRepairTime
              machine

     runProcessInStartTime machine
     runProcessInStartTime machine

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

     return $
       results
       [resultSource
        "upTimeProp"
        "The long-run proportion of up time (~ 0.66)"
        upTimeProp]
```

The point is that we can parameterize not only by the `IO` computation. Similarly, we could convert the sequential model to an equivalent model by using the distributed simulation computation, or nested simulation computation. It is important if we want to build a complex model from the sequential model prototype. We could test and validate the prototype and only then begin transforming it to the distributed model, for example.

11.4 Writing Generalized Code

To write the code, it is not necessary to instantiate always the generalized computations by some parameter computation such as `IO`. So, if you decide to add the `MonadDES` or `MonadSD` type class constraints to your function then it is highly recommended to add the `INLINE` or `INLINABLE` pragma too like this.

```
-- | Actuate the event handler in the specified time points.
enqueueEventWithTimes :: MonadDES m => [Double] -> Event m () -> Event m ()
{-# INLINABLE enqueueEventWithTimes #-}
enqueueEventWithTimes ts e = loop ts
  where loop []       = return ()
        loop (t : ts) = enqueueEvent t $ e >> loop ts
```

Without this pragma, the `enqueueEventWithTimes` function would be slow, because of using the `MonadDES` constraint.

At the same time, there would be no need to add such a pragma if your function would have no type class constraints, for example, when using the already instantiated simulation computations like `Event IO m`.

Chapter 12

Distributed Simulation Computation

Aivika distribution includes the `aivika-distributed` package that allows creating parallel and distributed simulation models[12] based on the optimistic Time Warp method[3].

We run *logical processes*, possibly on different computers, or on different threads of the same computer, or combining the both approaches. These processes send themselves asynchronous *messages* that have timestamps. The timestamp is a modeling time at which the specified message should be processed.

The main issue of distributed simulation is as follows. So called the *Paradox of Time* may happen, when some logical process may receive a message that would look to the past of that process, where the message would have to be processed in the past.

There are *conservative* methods that excludes the very possibility of occurring the Paradox of Time. On the contrary, there are also *optimistic* methods that do allow occurring the paradox, but they provide means for rolling the simulation back so that the problematic message would indeed be processed precisely at that time it has to be processed. We revert the modeling time of the logical process in case of need. This rollback can be cascade involving many or even all logical processes if required. The most famous optimistic method is called *Time Warp*, which Aivika implements.

To pass messages over the network, Aivika uses Cloud Haskell represented by package `distributed-process`[1] and such protocol implementations as `distributed-process-simplelocalnet`[2]. Probably, you will need to learn how to use these packages if you are going to use the Aivika distributed simulation module.

[1] https://hackage.haskell.org/package/distributed-process

[2] https://hackage.haskell.org/package/distributed-process-simplelocalnet

12.1 DIO Computation

The `aivika-distributed` package exports a module that defines the `DIO` distributed simulation computation. This computation allows creating parallel and distributed simulation models.

```
module Simulation.Aivika.Distributed

data DIO a

instance MonadDES DIO
```

Below the `ProcessId` type is used from the `distributed-process` package. In our case it identifies a logical process. It is not important where that process actually resides. It can be launched in another thread of the same operating system process, or in another process of the same physical computer, or it can represent a logical process that runs on another remote computer. The `ProcessId` identifier transparently represents the logical process in all these cases.

To distinguish this type from the discontinuous process identifiers, we'll prefix it by the `DP` namespace that should correspond to a qualified import from package `distributed-process`.

```
import qualified Control.Distributed.Process as DP
import Control.Distributed.Process.Serializable
```

To send the message to another logical process, there is the following function.

```
sendMessage :: forall a. Serializable a => DP.ProcessId -> a -> Event DIO ()
```

Here the `Serializable` type class denotes something that can be serialized to binary data and then sent over the network. Please consult the documentation of the `distributed-process` package.

The receive time will equal to the send time in the `sendMessage` function. The original Time Warp method is able to process rollbacks in such cases. But Aivika also supports rollbacks of additional kind that arise in case of retrying the computations, for example, when the logical process comes temporarily to illegal state, because not all messages have been delivered yet. Then we cannot proceed with the simulation right now and we have to wait for other messages. This situation is described in section 12.7.

Because of those rollbacks of the additional kind, it is highly recommended to not use the `sendMessage` function at all; otherwise, the distributed simulation may fall into infinite loop. The receive time should be greater than the send time. You can add one millionth to the current modeling time and it will work. Perhaps in one of the future versions of Aivika this restriction will be resolved.

To specify the exact receive time at which the message should be handled by another logical process, we can use the next function.

```
enqueueMessage ::
    forall a. Serializable a => DP.ProcessId -> Double -> a -> Event DIO ()
```

Each time the logical process receives a message, the corresponding signal is triggered at the receive time by passing the message itself in signal value to all listeners.

```
messageReceived :: forall a. Serializable a => Signal DIO a
```

If we want our logical process will receive the incoming messages of the specified type then we have to subscribe for handling this signal.

12.2 Running DIO Computation and Time Server

Every start of the `DIO` computation is equivalent to a start of the corresponding logical process. It is obvious that such simulation computations as `Event DIO a` or `Process DIO a` can ultimately be reduced to the `DIO` computation. Then the question is how to run the `DIO` computation itself?

Before doing this, we have to introduce the notion of so called *time server*. At least, this is how it is called in Aivika and this name can be not very common and widespread in the literature.

Aivika uses Samadi's algorithm[2, 12] to synchronize the *global virtual time* among all logical processes of the distributed simulation. The mentioned time server is responsible for performing such a synchronization. Therefore, every run of the distributed simulation must start by launching a new time server, which should be single and unique for the whole distributed simulation.

```
timeServer :: Int -> TimeServerParams -> DP.Process ()
```

The first parameter defines a quorum for logical processes. After the specified number of logical processes connect to the time server, this time server begins synchronizing the global virtual time. The second parameter specifies the time server parameters. In the examples we'll use the default parameters, but you should consult the `aivika-distributed` documentation for more detail.

```
defaultTimeServerParams :: TimeServerParams
```

Since the time server is designed to be launched in a separate node of the cluster, there is a helper function that simplifies this use case:

```
curryTimeServer :: (Int, TimeServerParams) -> DP.Process ()
```

Note that `DP.Process ()` is not the same as `Process ()`. The former is a computation defined in the `distributed-process` package, while the latter represents some discontinuous process within simulation.

So, after running somewhere a single instance of the time server for the entire distributed simulation, we receive the time server process identifier `DP.ProcessId` and start running logical processes one by one, either in different local or remote nodes of the cluster.

Here the most simple strategy is to use one *master* node and many *slave* nodes. Before running the master, we start the slave nodes, probably, on different computers. Only then we run the master that connects to the slaves and a new distribution simulation begins. After the simulation finishes, the master node disconnects from the slave nodes, then it returns a final result and stops, while the slave nodes can continue their work by waiting for a new connection from new master node to run a new distributed simulation. Please

consult the `distributed-process-simplelocalnet` documentation to better understand how it works.

To run a new logical process, we have to pass the time server process identifier in the arguments of the following function.

```
runDIO :: DIO a -> DIOParams -> DP.ProcessId
          -> DP.Process (DP.ProcessId, DP.Process a)
```

The first parameter defines the corresponding simulation computation that we reduced to the `DIO` computation preliminarily. The second parameter defines the logical process parameters, but we'll use the default parameters specified by value `defaultDIOParams` for simplicity.

```
defaultDIOParams :: DIOParams
```

The `runDIO` function launches a helper process that can accept incoming messages from other logical processes and pass them to the corresponding simulation computation. That helper process is called *inbox*. The inbox process identifier and the simulation computation process are returned by this function. Please note that it does not start the underlying simulation computation yet. We have to apply and involve the returned simulation computation process explicitly. Such an approach is quite flexible to allow us to run the distributed simulation in different configurations.

To register itself in the time server, the logical process must apply the `registerDIO` function. If the quorum is satisfied then the time server begins synchronization.

```
registerDIO :: DIO ()
```

After the simulation finishes, the logical process of the master node should call the `terminateDIO` function, while a similar logical process on the slave node should call the `unregisterDIO` function.

```
terminateDIO :: DIO ()
unregisterDIO :: DIO ()
```

By the way, we might run a few distributed simulations on the same cluster. There would be the same number of time server instances. Different logical processes would share the same cluster nodes. But this scenario can be impractical as those simulations would interferer with each other by concurrent accessing to the same limited computer resources.

12.3 Example: Equivalent Sequential Simulation

To demonstrate how the distributed simulation can be launched, we use the same model from section 11.3. The model itself is still sequential formally, but it is actually rewritten as distributed. The difference from the true distributed simulation is that we use one node only. Everything else are attributes of the distributed simulation.

```
import Control.Monad
import Control.Monad.Trans
import Control.Concurrent
import qualified Control.Distributed.Process as DP
import Control.Distributed.Process.Node (initRemoteTable)
import Control.Distributed.Process.Backend.SimpleLocalnet

import Simulation.Aivika.Trans
import Simulation.Aivika.Distributed

meanUpTime = 1.0
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 10000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation DIO ()
model =
  do totalUpTime <- newRef 0.0

     let machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)
              repairTime <-
                randomExponentialProcess meanRepairTime
              machine

     runProcessInStartTime machine
     runProcessInStartTime machine

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

     let rs =
           results
           [resultSource
            "upTimeProp"
            "The long-run proportion of up time (~ 0.66)"
            upTimeProp]

     printResultsInStopTime printResultSourceInEnglish rs

runModel :: DP.ProcessId -> DP.Process ()
runModel timeServerId =
  do DP.say "Started simulating..."
     let ps = defaultDIOParams { dioLoggingPriority = NOTICE }
         m =
           do registerDIO
              a <- runSimulation model specs
              terminateDIO
              return a
     (modelId, modelProcess) <- runDIO m ps timeServerId
     modelProcess

master = \backend nodes ->
```

```
  do liftIO . putStrLn $ "Slaves: " ++ show nodes
     let timeServerParams =
           defaultTimeServerParams { tsLoggingPriority = DEBUG }
     timeServerId  <-
       DP.spawnLocal $ timeServer 1 timeServerParams
     runModel timeServerId

main :: IO ()
main = do
  backend <- initializeBackend "localhost" "8080" rtable
  startMaster backend (master backend)
    where
      rtable :: DP.RemoteTable
      -- rtable = __remoteTable initRemoteTable
      rtable = initRemoteTable
```

Note to the ceremony we need to run the master node. This is actually a template suitable for running logical processes on the slave nodes too. We'll use it further. Please see the `distributed-process-simplelocalnet` documentation to understand this pattern.

Also here we added different log levels by specifying the time server and logical process parameters accordingly.

When running the model in WinGCHi on Windows 7, I received the following output, although slightly edited manually with removed timestamps and process identifiers to make it shorter.

```
GHCi, version 8.2.1: http://www.haskell.org/ghc/  :? for help
Prelude> :cd C:\Docs\Tests\test02-aivika-book
Prelude> :load "MachRep1Simple.hs"
[1 of 1] Compiling Main             ( MachRep1Simple.hs, interpreted )
Ok, 1 module loaded.
*Main> main
Slaves: []
[INFO] Time Server: starting...
Started simulating...
[DEBUG] Time Server: RegisterLogicalProcessMessage pid://localhost:8080:0:10
[INFO] Time Server: starting
[DEBUG] Time Server: computing the global time...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://localhost:8080:0:10 224.48176784549443
[INFO] Time Server: providing the global time = Just 224.48176784549443
[DEBUG] Time Server: computing the global time...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://localhost:8080:0:10 10000.0
[INFO] Time Server: providing the global time = Just 10000.0
[DEBUG] Time Server: TerminateTimeServerMessage pid://localhost:8080:0:10
[INFO] Time Server: start terminating...
[INFO] Time Server: terminate
[ERROR] Exception occured: ProcessTerminationException
----------

-- simulation time
t = 10000.0

-- The long-run proportion of up time (~ 0.66)
upTimeProp = 0.6707069678156755
```

If you will try to run the model in the Terminal on macOS then some lines could be messed up. It is related to concurrent output to the same terminal window.

It is worth noting that the `ProcessTerminationException` is not actually error. It denotes that the corresponding inbox process or time server process terminates. Here we could see from zero to two such messages generated by a single logical process and/or the time server process. This is just a coincidence that we see exactly one message, because the master process had stopped earlier. By the tradition widely applied in server programming, this message has the `ERROR` log level. In all other cases `ERROR` would indeed mean an error, but not now. It was made intentionally to simplify the monitoring of the distributed simulation.

12.4 Example: Making Simulation Distributed

In this section it will be shown how the model from section 12.3 can be converted to a distributed model with message passing. This is quite an artificial example. From the point of simulation, the example is even useless and the resulting model is much slower than the original by obvious reasons, but like many books this text is focused on describing the tools that can be useful for you to build real models. So, this section is devoted to the functions for message passing as well as to that how you can run a cluster.

We define a counter in the master node but send updates from two slave nodes. Each slave node represents a separate machine tool. Therefore, the master node must subscribe to receiving these updates. It is obvious that the model creates a plenty of superfluous messages, but it demonstrates how the messages can be sent and then handled by another logical process.

Since here are no computation retries, we can still use the `sendMessage` function for simplicity, but you should understand that it is risky. Please use the `enqueueMessage` function with a small time gap whenever possible.

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DeriveDataTypeable #-}

import System.Environment (getArgs)

import Data.Typeable
import Data.Binary

import GHC.Generics

import Control.Monad
import Control.Monad.Trans
import Control.Concurrent
import qualified Control.Distributed.Process as DP
import Control.Distributed.Process.Closure
import Control.Distributed.Process.Node (initRemoteTable)
import Control.Distributed.Process.Backend.SimpleLocalnet

import Simulation.Aivika.Trans
import Simulation.Aivika.Distributed

meanUpTime = 1.0
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 10000.0,
```

```
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

newtype TotalUpTimeChange =
  TotalUpTimeChange { runTotalUpTimeChange :: Double }
  deriving (Eq, Ord, Show, Typeable, Generic)

instance Binary TotalUpTimeChange

-- | A sub-model.
slaveModel :: DP.ProcessId -> Simulation DIO ()
slaveModel masterId =
  do let machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                sendMessage masterId (TotalUpTimeChange upTime)
              repairTime <-
                randomExponentialProcess meanRepairTime
              machine

     runProcessInStartTime machine

     runEventInStopTime $
       return ()

-- | The main model.
masterModel :: Int -> Simulation DIO ()
masterModel n =
  do totalUpTime <- newRef 0.0

     let totalUpTimeChanged :: Signal DIO TotalUpTimeChange
         totalUpTimeChanged = messageReceived

     runEventInStartTime $
       handleSignal totalUpTimeChanged $ \x ->
       modifyRef totalUpTime (+ runTotalUpTimeChange x)

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (fromIntegral n * y)

     let rs =
           results
           [resultSource
            "upTimeProp"
            "The long-run proportion of up time (~ 0.66)"
            upTimeProp]

     printResultsInStopTime printResultSourceInEnglish rs

runSlaveModel :: (DP.ProcessId, DP.ProcessId)
                 -> DP.Process (DP.ProcessId, DP.Process ())
runSlaveModel (timeServerId, masterId) =
  runDIO m ps timeServerId
  where
    ps = defaultDIOParams { dioLoggingPriority = NOTICE }
    m  = do registerDIO
            runSimulation (slaveModel masterId) specs
            unregisterDIO
```

```
startSlaveModel :: (DP.ProcessId, DP.ProcessId) -> DP.Process ()
startSlaveModel x@(timeServerId, masterId) =
  do (slaveId, slaveProcess) <- runSlaveModel x
     slaveProcess

runMasterModel :: DP.ProcessId
                  -> Int
                  -> DP.Process (DP.ProcessId, DP.Process ())
runMasterModel timeServerId n =
  runDIO m ps timeServerId
  where
    ps = defaultDIOParams { dioLoggingPriority = NOTICE }
    m  = do registerDIO
            a <- runSimulation (masterModel n) specs
            terminateDIO
            return a

remotable ['startSlaveModel, 'curryTimeServer]

master = \backend nodes ->
  do liftIO . putStrLn $ "Slaves: " ++ show nodes
     let [n0, n1, n2] = nodes
         timeServerParams =
           defaultTimeServerParams { tsLoggingPriority = DEBUG }
     timeServerId <-
       DP.spawn n0
       ($(mkClosure 'curryTimeServer) (3 :: Int, timeServerParams))
     (masterId, masterProcess) <- runMasterModel timeServerId 2
     forM_ [n1, n2] $ \node ->
          DP.spawn node
          ($(mkClosure 'startSlaveModel) (timeServerId, masterId))
     masterProcess

main :: IO ()
main = do
  args <- getArgs
  case args of
    ["master", host, port] -> do
      backend <- initializeBackend host port rtable
      startMaster backend (master backend)
    ["slave", host, port] -> do
      backend <- initializeBackend host port rtable
      startSlave backend
  where
    rtable :: DP.RemoteTable
    rtable = __remoteTable initRemoteTable
```

We define a separate Haskell type for each kind of messages and make it an instance of the `Serializable` type class. For that, we use an ability of GHC to derive the type class instances.

Also we invoke the functions on remote nodes. Cloud Haskell helps us to do this, but we have to enable the Template Haskell extension.

Provided that the model was saved in the *MachRep1.hs* file, we can compile it by typing the following command in the terminal.

```
$ ghc -O2 -threaded MachRep1.hs
```

The distributed model is supposed to be run on four nodes: three slaves and one master node. In the beginning, we start our three slave nodes. For

simplicity, we run all nodes on localhost that has IP address 127.0.0.1, but we could specify other IP addresses from local net.

```
$ ./MachRep1 slave 127.0.0.1 8080 &
$ ./MachRep1 slave 127.0.0.1 8081 &
$ ./MachRep1 slave 127.0.0.1 8082 &
```

The represented model indeed assumes that all nodes exist in the local net and the master node has to find them without additional configuration. But in real case we could rewrite the model and put the actual IP addresses in some configuration file so that the master node could read that file and then establish the connections with remote nodes. Then there would be no restriction to run all nodes in the local net. We could fully use the Internet, instead.

Now to run the entire distributed simulation, we launch the master node that initiates all the work.

```
$ ./MachRep1 master 127.0.0.1 8088
```

In my case I received the next output, provided here with some truncation to make it shorter:

```
Slaves: [nid://127.0.0.1:8080:0,nid://127.0.0.1:8081:0,nid://127.0.0.1:8082:0]
[INFO] Time Server: starting...
[DEBUG] Time Server: RegisterLogicalProcessMessage pid://127.0.0.1:8081:0:10
[DEBUG] Time Server: RegisterLogicalProcessMessage pid://127.0.0.1:8082:0:10
[DEBUG] Time Server: RegisterLogicalProcessMessage pid://127.0.0.1:8088:0:9
[INFO] Time Server: starting
[DEBUG] Time Server: computing the global time...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8088:0:9 10000.0
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8082:0:10 0.7354694026911183
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8081:0:10 0.2316644765668638
[INFO] Time Server: providing the global time = Just 0.2316644765668638
[DEBUG] Time Server: computing the global time...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8082:0:10 408.2912566906416
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8081:0:10 453.81097898067776
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8088:0:9 897.564122712003
[INFO] Time Server: providing the global time = Just 408.2912566906416
[DEBUG] Time Server: computing the global time...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8082:0:10 10000.0
----------

-- simulation time
t = 10000.0

-- The long-run proportion of up time (~ 0.66)
upTimeProp = 0.6679609105729915

[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8088:0:9 10000.0
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8081:0:10 10000.0
[INFO] Time Server: providing the global time = Just 10000.0
[DEBUG] Time Server: UnregisterLogicalProcessMessage pid://127.0.0.1:8082:0:10
```

```
[INFO] Time Server: providing the global time = Just 10000.0
[DEBUG] Time Server: UnregisterLogicalProcessMessage pid://127.0.0.1:8081:0:10
[INFO] Time Server: providing the global time = Just 10000.0
[ERROR] Exception occurred: ProcessTerminationException
[ERROR] Exception occurred: ProcessTerminationException
[DEBUG] Time Server: TerminateTimeServerMessage pid://127.0.0.1:8088:0:9
[INFO] Time Server: start terminating...
[INFO] Time Server: terminate
[ERROR] Exception occurred: ProcessTerminationException
```

In this dump we see how the global virtual time increased monothonically until it became equal to the final simulation time.

If you are a happy user of Linux or macOS then you can start the master node many times, each time running a new distributed simulation with the same slave nodes. Unfortunately, by some unknown cause the network libraries work unstable on Windows...

Finally, it is worth noting that in the recent versions of the applied network libraries word `localhost` is not recognized as an acceptable IP address anymore. Instead, you should specify explicitly 127.0.0.1.

12.5 Input/Output Operations

The Input/Output operations require a special treating in the optimistic distributed simulation, for they cannot be reverted in general case. If such a request comes from the model then the corresponding logical process tries to synchronize its value of the global virtual time. It will wait until the local queue time will become equal to the global virtual time. The difficult question is how to handle a possible situation if the rollback occurs in the course of this synchronization. Fortunately, there is a robust and flexible way how to process safely such a rollback, which we'll consider below in this section.

The distributed `Event` and `Process` computations are `MonadIO` instances.

```
instance MonadIO (Event DIO)
instance MonadIO (Process DIO)
instance MonadIO (Composite DIO)
```

We might call directly the `liftIO` function from any place of the distributed simulation, but it would be unsafe by the reason described above. If the rollback indeed occurred during the global virtual time synchronization and the local queue time was still greater than the global virtual time, then a run-time error would be raised that should lead to interrupting the entire simulation. The solution could be to use `liftIO` only in safe blocks.

There is a very similar analog of the `enqueueEvent` function that also registers a new event at the specified time.

```
enqueueEventIO :: EventIOQueueing m => Double -> Event m () -> Event m ()
```

The key difference is that it guarantees that the local queue time will be equal to the global virtual time, when invoking the corresponding event handler at the specified time. This is exactly what we need to call safely `liftIO`!

```
instance EventIOQueueing DIO
```

How does it work? The `enqueueEventIO` function places the new event in the queue too. But before actuating the corresponding event handler, the logical process begins synchronizing its value of the global virtual time. It waits until the local queue time becomes equal to the global virtual time. If the condition satisfies then the event handler is invoked, where we can safely call `liftIO` as the local time is already synchronized.

But if the rollback occurs instead of successful synchronization then nothing special is applied. It just interrupts the synchronization and the rollback takes place. Nothing has changed, none `IO` action has been applied. So, it can safely perform the rollback. After the rollback, the corresponding event can be repeated again after its time comes with the next attempt.

So, it is quite safe to write the code like this:

```
t <- liftDynamics time
liftEvent $
  enqueueEventIO t $
  liftIO $
  -- here we could read in or write to some file, for example
```

Actually, the `Process` computation could implicitly generate such a code, when applying the `liftIO` function. But it was decided to not implement this as the order of `IO` actions would be unpredictable then.

Finally, because of permanent synchronizations of `IO` actions, it is highly recommended to use the `Ref` reference instead of the standard `IORef` reference whenever possible. The `Ref` type is much, much more lightweight and fast. Moreover, the `Ref` reference efficiently supports rollbacks.

12.6 Modeling Time Horizon

There is one unexpected use case for the `enqueueEventIO` function. With help of it we can limit the *horizon of modeling time* during distributed simulation, which specifies how far we can go from some time point in the past to which we can rollback yet.

```
runEventInStartTime $
  enqueueEventIOWithTimes [100, 200..] $
  return ()
```

Here the local queue time will be synchronized with the global virtual time in time points 100, 200, 300, ... Whether this is a good idea or not depends on the model. It can have a negative impact on the performance, but it can decrease the number of rollbacks simultaneously.

Regarding the modeling time horizon, the `DIOParams` type specifies thresholds for some queue sizes. Besides other queues, there is a log of actions and a queue for transient messages, where the logical process sent the message but did not receive yet an acknowledgment from another logical process.

There are defaults for these threshold values. We can change them, before starting a new distributed simulation. So, if one of the thresholds is exceeded then the logical process passes to the throttling mode, when it processes only the past messages, trying to decrease the queue sizes.

The global virtual time value has a very important meaning in the optimistic distributed simulation. We cannot rollback to the time, which is less than the

global virtual time. Therefore, we can safely remove all items from the queues that are related to the past with time values, which are less than the global virtual time.

12.7 Retrying Computations

The distributed simulation has another caveat related to the fact that not all messages may arrive in time or in order. As a result, such a strange situation may arise when the logical process passes temporarily to illegal state and the further proceeding with simulation has no sense until we receive all messages, which should lead to a saving rollback that would already recover the normal order of simulation.

For example, we cannot release a resource or GPSS storage if it already has the maximum available contents. If such a situation still happen then a special exception of type `SimulationRetry` is raised, which is handled by the Aivika distributed module. The simulator passes to a special mode, when it accepts the messages only in hope that they will eventually lead to a rollback.

We can artificially raise this exception by calling the next function that accepts a debugging message that will be displayed in the terminal window if the attempt to retry the computation will still fail.

```
retryEvent :: MonadException m => String -> Event m a
```

There is one caution related to the rollbacks of such a second kind, though. If one of the messages has equaled receive and send time then the simulation may fall into infinite loop. Probably, this is just a limitation of the current implementation.

At least, if the receive time will be greater than the send time, had their difference been one millionth, then the rollbacks of this second kind are handled properly. Therefore, it is strongly recommended to use the `enqueueMessage` function instead of `sendMessage`.

12.8 Recovering after Temporary Connection Errors

Aivika is able to recover the distributed simulation after temporary connection errors, but this mode must be enabled explicitly in the configuration parameters.

For example, if we'll take a model from 12.4 then we'll need to apply the following changes:

```
runSlaveModel :: (DP.ProcessId, DP.ProcessId)
                 -> DP.Process (DP.ProcessId, DP.Process ())
runSlaveModel (timeServerId, masterId) =
  runDIO m ps timeServerId
  where
    ps = defaultDIOParams { dioLoggingPriority = NOTICE,
                            dioProcessMonitoringEnabled = True,
                            dioProcessReconnectingEnabled = True }
    m  = do registerDIO
            runSimulation (slaveModel masterId) specs
            unregisterDIO

startSlaveModel :: (DP.ProcessId, DP.ProcessId) -> DP.Process ()
```

```
startSlaveModel x@(timeServerId, masterId) =
  do (slaveId, slaveProcess) <- runSlaveModel x
     DP.send slaveId (MonitorProcessMessage timeServerId)
     DP.send masterId (MonitorProcessMessage slaveId)
     DP.send slaveId (MonitorProcessMessage masterId)
     slaveProcess

runMasterModel :: DP.ProcessId
                  -> Int
                  -> DP.Process (DP.ProcessId, DP.Process ())
runMasterModel timeServerId n =
  runDIO m ps timeServerId
  where
    ps = defaultDIOParams { dioLoggingPriority = NOTICE,
                            dioProcessMonitoringEnabled = True,
                            dioProcessReconnectingEnabled = True }
    m  = do registerDIO
            a <- runSimulation (masterModel n) specs
            terminateDIO
            return a

remotable ['startSlaveModel, 'curryTimeServer]

master = \backend nodes ->
  do liftIO . putStrLn $ "Slaves: " ++ show nodes
     let [n0, n1, n2] = nodes
         timeServerParams =
           defaultTimeServerParams { tsLoggingPriority = DEBUG,
                                     tsProcessMonitoringEnabled = True,
                                     tsProcessReconnectingEnabled = True }
     timeServerId <-
       DP.spawn n0
       ($(mkClosure 'curryTimeServer) (3 :: Int, timeServerParams))
     (masterId, masterProcess) <- runMasterModel timeServerId 2
     DP.send masterId (MonitorProcessMessage timeServerId)
     forM_ [n1, n2] $ \node ->
          DP.spawn node
          ($(mkClosure 'startSlaveModel) (timeServerId, masterId))
     masterProcess
```

We enable the `dioProcessMonitoringEnabled` parameter along with the `dioProcessReconnectingEnabled` parameter for the logical processes. Similarly, we enable the `tsProcessMonitoringEnabled` parameter as well as the `tsProcessReconnectingEnabled` parameter for the time server. Also we send the `MonitorProcessMessage` message to start sending keep-alive messages for each possible connection by specifying the sender and receiver process identifiers, respectively.

In such a case, the distributed simulation is able to recover itself after temporary connection errors. But the simulation cannot wait indefinitely long for other logical processes. What if one of the cluster nodes shut down completely? You will find an answer in the next section.

12.9 Stopping Disconnected Simulation

When disconnection errors occur, there are time-out intervals within which the distributed simulation will wait for other logical processes before considering them as completely lost. It is required to stop automatically the distributed

simulation if the cluster breaks for a long enough time by some reason. One of the cluster nodes may reset or shut down. The network connection can be lost forever and so on.

The approach is quite simple. Since the time server is a single point of failure, it is sufficient to check the connections of the time server, which is unique for each distributed simulation run.

By default strategy, if the time server did not receive any message from one of the logical processes within the time-out interval specified by data constructor `TerminateDueToLogicalProcessTimeout` (=5 minutes by default) then the time server stops itself. To this interval, we usually have to add also a time-out used for session synchronization specified by the `tsTimeSyncTimeout` parameter (=1 minute by default). So, after 6 minutes (=5+1) the time sever shall stop itself.

There are similar data constructor `TerminateDueToTimeServerTimeout` and parameter `dioSyncTimeout` for the logical process with the same default values. They mean that if the logical process did not receive any message from the time server within the specified time-out interval then the logical process stops itself. By default, it will stop after 6 minutes (=5+1) too.

In other words, if the cluster is broken then its alive rest part will stop itself, at least, in 12 minutes (=6+6). But you can change the default values.

Therefore, it is very important to send keep-alive messages that must be enabled by direct sending the `MonitorProcessMessage` message in the very beginning of every distributed simulation run. Then you should also enable the monitoring and reconnecting parameters as it was described in section 12.8.

12.10 Distributed Simulation as Service

In continuation of the previous two sections, we'll talk about the creation of simulation services.

So, the distributed simulation can recover itself after temporary connection errors. But if the disconnection lasts too long then all parts of the distributed simulation cluster automatically stop themselves after exceeding the time-out intervals that specify how long we can wait for logical processes in case of connection errors.

Even if the slave logical process stops then its computational node remains active and it can accept a new simulation run. But if the node's computer was restarted then the node itself can be launched again, for example, as a service of the Linux operating system, and the node becomes ready again to accept the new simulation run. Something serious must happen if the node cannot restore itself, but it is usually solved on another level by duplicating the corresponding computer node.

Regarding the master logical process, it works by principle *all-or-nothing*. Either the master logical process will return the result, and it will be a correct result, for the simulation is analytic, or the master logical process will exit with some error. In the last case we can start a new simulation run after some delay in hope that the slave nodes will restore.

All this allows creating safe and robust simulation services with the desired response characteristics based on using the Linux operation system for the distributed cluster.

12.11 Monitoring Distributed Simulation

We might monitor the distributed simulation by observing the log messages, but it is more reasonable and preferable to write errors and warnings only in the log, while the monitoring itself can be implemented on another level.

There are additional run functions for the time server and logical process. They allow us to specify the handlers that would already process the monitoring information, probably, by sending this information to some external specialized tool.

```
timeServerWithEnv :: Int -> TimeServerParams -> TimeServerEnv -> DP.Process ()
```

This function runs a new time server, where the `TimeServerEnv` data type can specify the function that will be called from a separate process each time the Aivika simulator wants to send the monitoring data.

```
data TimeServerEnv =
  TimeServerEnv { tsSimulationMonitoringAction ::
                    Maybe (TimeServerState -> DP.Process ())
                }
```

The `TimeServerState` data type describes the current state of the time server. It includes the global virtual time and a list of registered logical processes. In the handler, you can send this information further to a special monitoring tool, or just print it in the standard output, or write it to some file.

The interval between sessions of sending the monitoring data is specified by the `tsSimulationMonitoringInterval` parameter of `TimeServerParams` that you can define, when starting the time server.

Similarly, we can run a new logical process by specifying a parameter of the `DIOEnv` type:

```
runDIOWithEnv :: DIO a -> DIOParams -> DIOEnv -> DP.ProcessId
                 -> DP.Process (DP.ProcessId, DP.Process a)
```

This type can contain a handler too, but suitable already for processing the logical process monitoring information.

```
data DIOEnv =
  DIOEnv { dioSimulationMonitoringAction ::
             Maybe (LogicalProcessState -> DP.Process ())
         }
```

The `LogicalProcessState` data type describes the current state of the logical process. It includes the local time of the process, the local queue time, the event queue size, the rollback log size, the count of input messages, the count of output messages, the count of outgoing messages that are not delivered yet, the total count of rollbacks performed. In the handler, you can also send this information further to the external monitoring tool, or print it in the standard output, or write it to some file.

Also the interval between monitoring sessions is specified by parameter `dioSimulationMonitoringInterval` of `DIOParams` that you can define, when starting the logical process.

Thus, the simulator can provide us with the actual monitoring information about the current state of the distributed simulation.

12.12 Distributed Simulation Experiment

As before, under the name of simulation experiment, we mean the Monte-Carlo simulation, where we create a Web page with the results of simulation represented as charts, histograms, summary statistics, links to generated CSV tables and so on.

The distributed simulation experiment is complicated by the fact that the optimistic method is applied, which means that a plenty of rollbacks can be generated in the course of simulation. Moreover, the simulation is distributed and the nodes of the cluster can reside in physically different computers.

Therefore, the following approach is suggested, which will work for any type of simulation, both sequential and distributed. The resulting data are processed in the end of every simulation run. The data are saved in SQL databases. After the simulation experiment is complete, we can build a report with the simulation results by using the data stored in the SQL databases.

We need additional libraries that are included in *Aivika Extension Pack*. See appendix A for installation instructions. But the subject goes beyond the scope of this book. So, the distributed simulation experiment is not described here. We note only that there is such a possibility.

12.13 Summary

Aivika allows running parallel and distributed discrete event simulation models based on using the optimistic Time Warp method. The simulation can be launched on a single computer, or a cloud service such as Amazon, or a true distributed cluster. Different network protocols are supported by Cloud Haskell, which Aivika uses to connect logical processes. For example, the computers can be connected via the ordinary Internet.

The distributed simulation is able to recover itself after temporary connection errors, but there is an embedded mechanism that stops the simulation after exceeding time-out intervals. So, we can build robust simulation services with the desired response characteristics. Moreover, such a distributed simulation can be monitored in real time.

Besides all wonderful advantages that the distributed simulation can give to us, it is worth noting again that this type of simulation is not free. In the most of cases the sequential simulation will be much faster and it will be much more easy-to-use. The message passing and a start of distributed simulation are costly operations. The start is related to a slow initialization of network services, while the message passing is costly per se, but it also may potentially generate rollbacks. So, there must be a strong reason, why the distributed model cannot be written as a sequential one. Either the model is too huge so that it cannot reside in the memory of single computer, or we can gain real benefits from parallelism by launching many parallel logical processes.

However, this is an exciting thing that we can build sequential and distributed simulation models based on the same unified approach that Aivika provides. As we'll see soon later, the same approach is suitable for nested simulation too.

Part III

Nested Simulation

This part is about the nested simulation, when we can run simulations within simulations, then new simulations within those simulations within simulations and so on, recursively. It can be divided into two types of nested simulation.

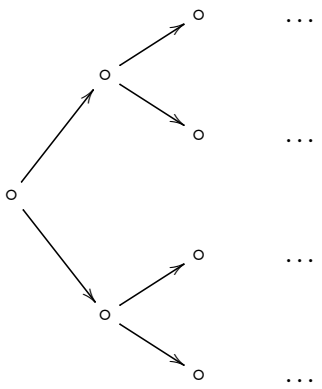
The first type suggests a more flexible approach, when we can branch a new nested simulation at any time, but it may lead to an exponential growth of simulations within the initial simulation. Therefore, we usually have to limit the growth of the tree of simulations by some depth.

The second approach uses a special but very smart structure instead of the tree. It is known as a lattice. Here the nested simulations are created only in a limited number of the lattice nodes, but we can traverse such nodes in quadratic time, though. For example, it could be useful for financial modeling.

Chapter 13

Exponential Branching

At first we'll consider a type of the nested simulation, when we can branch a new nested simulation at any time. It corresponds to a tree of simulations that potentially has an exponential growth of the nodes.



13.1 Branching Simulation Computation

The `aivika-branches` package implements the corresponding type of nested simulation. It defines the `BR` simulation computation, which should be parameterized itself by the `IO` monad to be an instance of `MonadDES` from chapter 11.

```
module Simulation.Aivika.Branch

data BR m a

instance MonadDES (BR IO)
```

The computation has a very simple run function that returns a value in the underlying monad.

```
runBR :: MonadIO m => BR m a -> m a
```

We'll see soon how we can create a tree of nested simulations. We have to limit this tree by some depth. Therefore, we need a function that would return the current branch level starting from 0.

```
branchLevel :: Monad m => BR m Int
```

The root source simulation has level 0. The next simulations created within the root simulation will have level 1. Then new nested simulations created within those next simulations will have level 2 and so on.

Now it is time to show how we can create nested simulations.

```
branchEvent :: Event (BR IO) a -> Event (BR IO) a
```

The `branchEvent` function branches a new nested simulation, runs the specified computation within it at the same modeling time and then returns the result to the parent simulation.

It is very important that the nested simulation cannot change in any way neither the event queue of the parent simulation, nor any its `Ref` reference values, that is, the nested simulation cannot change the state of the model from the parent simulation.

This is yet one reason, why you should use the `Ref` type instead of the standard `IORef` reference whenever possible. Although `IORef` can still be useful, for example, to turn some trigger on, if we have found something in the nested simulation, but the number of such scenarios is quite limited.

You can think of the new branch as a clone of the model state from the current simulation. This is a very cheap and fast operation, but it is not free, though. The clone inherits the recent state of its parent. The events that had to be actuated should be actuated too but already within the nested simulation without affecting the parent simulation. The `Ref` reference will inherit its state from the parent simulation, but the change of the `Ref` reference won't affect the parent simulation in any way.

Furthermore, we can branch a new nested simulation within the already branched simulation. Each time we create a new nested simulation, we increase the branch level by one relative to the parent simulation.

It can be useful to start the branched simulation with some delay in modeling time so that all pending events would be processed within the nested simulation. Therefore, there is another function that allows specifying the exact time at which we should run the given computation, although this is conceptually very similar to `branchEvent`, just the start modeling time differs:

```
futureEvent :: Double -> Event (BR IO) a -> Event (BR IO) a
```

Here the first parameter specifies the time of activating a computation, but the result is returned to the parent simulation at the current modeling time as before. It is important that the parent simulation is then proceeded with the current modeling time. Therefore, this is the `Event` computation by the way, not `Process`.

13.2 Example: Simulation Branches

To demonstrate the use of nested simulation, we'll consider a quite conceptual example, where we'll estimate the random value by averaging its further es-

timations made in the future. As usual, we'll use our favourite model from section 11.3.

> There are two machines, which sometimes break down. Up time is exponentially distributed with mean 1.0, and repair time is exponentially distributed with mean 0.5. There are two repairpersons, so the two machines can be repaired simultaneously if they are down at the same time. Output is long-run proportion of up time. Should get value of about 0.66.

The Haskell model is as follows. It is quite simple.

```
import Control.Monad
import Control.Monad.Trans

import Simulation.Aivika.Trans
import Simulation.Aivika.Branch

meanUpTime = 1.0
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 1000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

maxLevel = 10

delta :: Int -> Parameter (BR IO) Double
delta n =
  do t0 <- liftParameter starttime
     t2 <- liftParameter stoptime
     return $ (t2 - t0) / fromIntegral n

forecast :: Event (BR IO) Double -> Event (BR IO) Double
forecast m =
  do let loop dt0 i =
           do level <- liftComp branchLevel
              if level <= maxLevel
                then do t  <- liftDynamics time
                        x1 <- futureEvent (t + dt0) $ loop dt0 (i - 1)
                        x2 <- futureEvent (t + dt0) $ loop dt0 (i + 1)
                        let x = (x1 + x2) / 2
                        x `seq` return x
                else m
     dt0 <- liftParameter $ delta maxLevel
     loop dt0 0

model :: Simulation (BR IO) (Results (BR IO))
model =
  do totalUpTime <- newRef 0.0

     let machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)
              repairTime <-
                randomExponentialProcess meanRepairTime
              machine
```

```
     runProcessInStartTime machine
     runProcessInStartTime machine

     let upTimeProp =
           do x <- readRef totalUpTime
              t <- liftDynamics time
              return $ x / (2 * t)

     upTimePropForecasted <-
       runEventInStartTime $
       forecast upTimeProp

     return $
       results
       [resultSource
        "upTimeProp"
        "The long-run proportion of up time (~ 0.66)"
        upTimeProp,
        --
        resultSource
        "upTimePropForecasted"
        "The forecasted long-run proportion of up time"
        (return upTimePropForecasted :: Event (BR IO) Double)]

main :: IO ()
main =
  runBR $
  printSimulationResultsInStopTime
  printResultSourceInEnglish
  model specs
```

Note how we estimate the `Event` computation by dividing it into two parts and then getting the average value in the helper `forecast` function. We do it recursively so that the tree depth won't exceed 10 levels starting from 0. Here we create $2^{10+1} - 1 = 2047$ simulations including the source root simulation itself.

Also the `upTimePropForecasted` variable is a pure value of type `Double`. Therefore, we wrap it in the `Event` computation to return as a result source.

When running the model, I received the following output:

```
$ time ./MachRep1
----------

-- simulation time
t = 1000.0

-- The long-run proportion of up time (~ 0.66)
upTimeProp = 0.6784040642495368

-- The forecasted long-run proportion of up time
upTimePropForecasted = 0.6636117393274816


real 0m3.845s
user 0m3.778s
sys 0m0.039s
```

As we can see, 2047 simulations lasted for about 4 seconds. The root source simulation passed all the modeling time interval from start to end, while the

nested simulations were short-term with intervals equaled to $(1000-0)/10 = 100$ time units. The forecasted estimation seems to be more precise than the first value.

Chapter 14

Quadratic Traverse of Lattice

Appendix A

Installing Aivika

The instructions below are given for the case when you use Stack[1], although the same libraries can be installed using Cabal. Please note that the versions used in the instructions correspond to the time of writing this document. The actual and updated version numbers can be found at the AivikaSoft website[2].

A.1 Using Open Source Libraries Only

When using the open source Aivika libraries only, you can add the following dependencies to the `extra-deps` section in your *stack.yaml* file:

```
- aivika-5.3.1
- aivika-transformers-5.3.1
- aivika-distributed-0.8
- aivika-experiment-5.1
- aivika-experiment-chart-5.1
- aivika-experiment-diagrams-5.1
- aivika-experiment-cairo-5.1
- aivika-realtime-0.3
- aivika-branches-0.3
- aivika-lattice-0.3
- aivika-gpss-0.4
```

Now these libraries will be available in your Stack project.

A.2 Using Aivika Extension Pack

Aivika Extension Pack is a set of additional Aivika libraries which use is governed by its own license[3]. In short, Aivika Extension Pack is free for educational and academic projects, but the commercial license must be purchased in other cases.

To include the libraries from Aivika Extension Pack in your Stack project, you should add the following direct locations to the `packages` section in your *stack.yaml* file like this:

[1] http://docs.haskellstack.org/
[2] http://aivikasoft.com/en/install.html
[3] http://aivikasoft.com/downloads/aivika-extension-pack/license.txt

```
- location:
   https://github.com/dsorokin/aivika-experiment-entity/archive/v0.4.tar.gz
  extra-dep: true
- location:
   https://github.com/dsorokin/aivika-experiment-entity-HDBC/archive/v0.4.tar.gz
  extra-dep: true
- location:
   https://github.com/dsorokin/aivika-experiment-provider/archive/v0.4.tar.gz
  extra-dep: true
- location:
   https://github.com/dsorokin/aivika-experiment-provider-distributed/archive/v0.2.tar.gz
  extra-dep: true
- location:
   https://github.com/dsorokin/aivika-experiment-report/archive/v0.4.tar.gz
  extra-dep: true
- '.'
```

Then you should define the corresponding dependencies in the `extra-deps` section of the same file:

```
- aivika-experiment-entity-0.4
- aivika-experiment-entity-HDBC-0.4
- aivika-experiment-provider-0.4
- aivika-experiment-provider-distributed-0.2
- aivika-experiment-report-0.4
```

Now you can use Aivika Extension Pack in your Stack project.

A.3 API Reference Documentation

After installing Aivika, you can build the API reference documentation:

```
$ stack haddock
```

Appendix B

Charting Backend

Bibliography

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., USA, 1985.

[2] Samdai B. *Distributed simulation, algorithms and performance analysis*. PhD thesis, University of California, Los Angeles, 1985.

[3] Jefferson D.R. and B. Beckman et al. The Time Warp operating systems. *11th Symposium on Operating Systems Principles*, 21:77–93, 1987.

[4] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

[5] John Hughes. Programming with arrows. In *Advanced Functional Programming*, pages 73–129, 2004.

[6] `iThink` Software. `http://www.iseesystems.com`, 2014. Accessed: 1-May-2014.

[7] Robert Macey and George Oster. `Berkeley Madonna` Software. `http://www.berkeleymadonna.com`, 2014. Accessed: 1-May-2014.

[8] Norm Matloff. Introduction to discrete-event simulation and the `SimPy` language. `http://simpy.readthedocs.org/en/latest/`, 2008. Accessed: 1-May-2014.

[9] Henrik Nilsson and Antony Courtney et al. `Yampa` Library, Version 0.9.5. `http://hackage.haskell.org/package/Yampa`, 2014. Accessed: 1-May-2014.

[10] Ross Paterson. A new notation for Arrows. In *In International Conference on Functional Programming*, ICFP '01, pages 229–240. ACM, 2001.

[11] A.A.B. Pritsker and J.J. O'Reilly. *Simulation with Visual SLAM and AweSim*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1999.

[12] Fujimoto R.M. *Parallel and Distributed Simulation Systems*. Wiley Interscience, 2000.

[13] Thomas Schriber. *Simulation using GPSS*. Wiley, 1974.

[14] `SimPy` Library. `http://simpy.readthedocs.org/en/latest/`, 2014. Accessed: 1-May-2014.

[15] AnyLogic Software. http://www.anylogic.com, 2014. Accessed: 1-May-2014.

[16] Ilya I. Trub. *An Object-oriented Modeling in C++*. Piter, Russia, 2006. (In Russian).

[17] Vensim Software. http://vensim.com, 2013. Accessed: 1-May-2014.

Index