An Introduction to Aivika Simulation Library

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

July 17, 2013

Contents

Chapter 1

Introduction

In 2009 in the course of my studying the functional programming I invented one approach of integrating the system of ordinary differential equations that used the standard Runge-Kutta and Euler's methods but did it in a slightly unusual way using the functional programming approach. Before that I had developed a visual simulation tool Simtegra MapSys[3][1] together with Dr. Zahed Sheikholeslami for the field of System Dynamics.

It has turned out that the new approach was not limited to the differential equations only. The approach can be applied to the Discrete Event Simulation (DES) and Agent-based Modeling too. My method can be applied to simulating the wide range of dynamic systems that evolve and change in time.

Initially, I created an F# library which I called Aivika[6]. Here Aivika is also a female Mari name pronounced with accent on the last syllable. Then I ported the library to Haskell and Scala[10].

The Haskell version is hosted on HackageDB as a collection of three interconnected packages[7, 8, 9] that provide facilities for developing the models and analyzing their results. The results can be prepared by Aivika in a form of tables and charts, which requires minimal effort from the modeler.

Below is described the Haskell version of my simulation library Aivika. It is more clear from the standpoint of maths and programming. Moreover, the Haskell version is the most mature and it was tested on different known models. Its API has been mostly stabilized. It works on Linux, Windows and OS X.

In chapter 2 two monads `Dynamics` and `Simulation` are introduced. They are a key point of my approach. A computation within the `Dynamics` monad can be identified with some dynamic process that changes in time points. The `Simulation` monad already describes some computation that occurs only once within the simulation run, for example, when an integral is created, or the external parameter value is updated. Such a parameter can be either random, or it can be defined according to some design of the simulation experiment, or read from the file. It is important here that we can bind different computations

---

[1]The site is not accessible any more but the software is freely available by the link from http://www.systemswiki.org in the Simulation Software section.

to create new ones. So, it is shown how the recursive differential equations with loopbacks can be defined with help of these two monads and then simulated.

Chapter 3 introduces the Event Queue. The queue behaves like a coordination center processing the events. It is important that the event handlers are the `Dynamics` computations, namely some actions specified at time points, which binds the event processing with the main simulation. It allows us to simulate the models under the event-oriented paradigm of DES.

Chapter 4 develops the idea of the `Dynamics` computation further. A new monad `Process` is introduced. Only now the `Process` computation can be identified with some discontinuous process. Such a process can suspend at any time and then resume later. It is important that any `Dynamics` computation can be embedded in the `Process` computation as well as the `Process` computation can be started within the `Dynamics` computation at any specified time point. It allows us to combine these computations and eventually allows us to simulate the models under the process-oriented paradigm of DES, where all the coordination is performed implicitly by the event queue.

Chapter 5 returns us to the `Dynamics` and `Simulation` monads. It shows how we can simulate the models under the activity-oriented paradigm of DES. Such an activity can be involved in the main simulation with help of the event queue too.

Chapter 6 shows how my approach can be extended to support the basic case of the agent-based modeling. Following the general line, the agent handlers are the `Dynamics` computations. It allows us to involve the agents in the main simulation.

In chapter 7 we return to the differential equations. A small example of the System Dynamics model is provided.

The last chapter 8 summarizes my approach and briefly describes some topics which were not covered in this document before but which I found important to develop complex models, although it is difficult to describe everything as the library becomes larger with every new version. However, the main idea remains always very simple.

Chapter 2

Dynamic Systems

A *dynamic system* evolves and changes in time. An example is the system of ordinary differential equations with help of which we can describe some model of System Dynamics. Each time we define a Discrete Event Simulation model, we also define a time varying dynamic system. Finally, in the Agent-based Modeling we define the agents, where their behavior actually obeys the rules of some dynamic system too. What unites all these cases is that the resulting system depends on the time factor.

In mathematics there is a notion of the *random process*. This is a generalization of the numeric function of time. The process can return arbitrary random values in time points.

Strictly in a context of the Haskell programming language I will use an abstract notion of the *dynamic process*. Arbitrary values can be returned by such a dynamic process, including the values that can be non-determined, i.e. depend on the `IO` monad, which is necessary to define the stochastic system.

2.1   Basic Simulation Monads

In the Aivika simulation library the dynamic process is represented as the `Dynamics` monad.

```
module Simulation.Aivika.Dynamics

data Dynamics a

instance Functor Dynamics
instance Monad Dynamics
instance MonadFix Dynamics
instance MonadIO Dynamics

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

So, any value of the `Dynamics` monad describes some dynamic process that varies in time. This process can return arbitrary values including numbers in time points. Moreover, we can construct mathematical expressions from such processes, for this monad can be an instance of the standard type classes `Num`, `Fractional` and `Floating`.

What makes it a monad is an ability to bind different processes into one compound process. It is possible due to the fact that the `Dynamics` monad is very similar to the standard `Reader` monad. We only pass in the current simulation time and other related parameters to every part of the imperative computation.

There are four primitives that allow us to receive the current values of the time parameters:

```
module Simulation.Aivika.Dynamics.Base

starttime :: Dynamics Double
stoptime :: Dynamics Double
dt :: Dynamics Double
time :: Dynamics Double
```

The `starttime` computation represents the initial time of the simulation. The `stoptime` computation gives us the information about the final time of the simulation. The `dt` computation returns the integration time step. This is a heritage of System Dynamics, where we have to define an integration method with help of which we are going to integrate the system of differential equations. Aivika is a hybrid framework that supports different simulation paradigms. Therefore we must know the integration method and its parameters to simulate the models of System Dynamics. Finally, the `time` built-in computation returns the current simulation time.

Having only these definitions, we can define simple dynamic processes and functions that operate on them:

```
sinWave :: Dynamics Double -> Dynamics Double -> Dynamics Double
sinWave a p = a * sin (2.0 * pi * time / p)

cosWave :: Dynamics Double -> Dynamics Double -> Dynamics Double
cosWave a p = a * cos (2.0 * pi * time / p)
```

Using the *do*-notation, we could achieve the same goal differently.

```
sinWave a p =
  do a' <- a
     p' <- p
     t' <- time
     return $ a' * sin (2.0 * pi * t' / p')
```

The dynamic process can return the integral value. But there is one difficulty with the integral. It must be created before and this action must occur only once within the simulation run.

To represent such actions, Aivika introduces the `Simulation` monad. It represents a value that doesn't change within the current simulation run, or

defines some action which occurs during this run. For example, the `Simulation` computation is ideal for representing the random external parameters for the Monte-Carlo simulation, or for representing the actions that create integrals. Like the `Dynamics` monad, the `Simulation` monad can be used for constructing expressions. Moreover, any `Simulation` computation can be *lifted* to the `Dynamics` computation.

```
module Simulation.Aivika.Dynamics.Simulation

data Simulation a

instance Functor Simulation
instance Monad Simulation
instance MonadFix Simulation
instance MonadIO Simulation

instance (Num a) => Num (Simulation a)
instance (Fractional a) => Fractional (Simulation a)
instance (Floating a) => Floating (Simulation a)
```

Here is the key function that creates an integral by the specified derivative and initial value.

```
module Simulation.Aivika.Dynamics.SystemDynamics

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

To create loopbacks in the recursive differential equations, you should use so called the recursive *do*-notation, which is possible due to the fact that the `Simulation` monad is an instance of the `MonadFix` type class.

Let us consider the following ODE system:

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

Its equivalent will take the following form in Aivika:

```
{-# LANGUAGE RecursiveDo #-}

...

model :: Simulation (Dynamics [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
      return $ sequence [time, a, b, c]
```

We can simulate this model in two stages. At first, we have to request for the values of the dynamic processes `time`, `a`, `b`, `c` in some time points. There is a plenty of such functions.

For simplicity, I will use that one which returns the value of the `Dynamics` computation in the final time point. The function returns the result within the `Simulation` computation, i.e. during the simulation run.

```
module Simulation.Aivika.Dynamics

runDynamicsInStopTime :: Dynamics a -> Simulation a
```

At second, we have to run the `Simulation` computation itself by specifying the specs that have the obvious meaning:

```
module Simulation.Aivika.Dynamics.Simulation

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

data Specs = Specs { spcStartTime :: Double,
                     spcStopTime :: Double,
                     spcDT :: Double,
                     spcMethod :: Method }
            deriving (Eq, Ord, Show)

data Method = Euler | RungeKutta2 | RungeKutta4
             deriving (Eq, Ord, Show)
```

The specs are namely that thing which provides the `starttime`, `stoptime`, `dt` and `time` built-in computations with the input data. The specified integration method has effect only on the integrals.

Let the initial time be 0, final time be 10, integration time step equal 0.001 and we apply the 4th order Runge-Kutta method.

```
specs = Specs { spcStartTime = 0,
                spcStopTime = 10,
                spcDT = 0.001,
                spcMethod = RungeKutta4 }
```

Now we can simulate our ODE system.

```
main =
  let s = model >>= runDynamicsInStopTime
  in runSimulation s specs >>= print
```

We will receive the following simulation results in the final time point:

```
[10.0,
 4.539992976248895e-3,
 4.53999297624868e-2,
 99.95006007726207]
```

2.2 Automating Simulation Experiments

The `Dynamics` and `Simulation` monads are quite flexible. They are not only monads but they allow using the recursive *do*-notation. Also they allow embedding any `IO` computation. It means that very many simulation models can be described in their language and then simulated using the corresponded run functions by the specified simulation specs. But direct returning every simulation variable and passing in it to the run functions is quite tedious. There is a more simple solution, though.

The Aivika library has additional packages Aivika Experiment[8] and Aivika Experiment Chart[9]. They are hosted on the HackageDB site as well as Aivika itself[7]. These two packages allow automating the simulation experiments. The packages were tested on Linux, Windows and OS X. What is important, they use only public API of Aivika. They demonstrate the potential of the Aivika simulation library.

The idea is as follows. There is type `ExperimentData` that contains all the necessary information about the model variables to be analyzed. Also there are `String` names of these variables. The modeler must create a model that just returns a value of type `Simulation ExperimentData`. In case of need, the model can be even closed-source.

```
module Simulation.Aivika.Experiment

data ExperimentData
data Experiment
```

Then the analyst, who can be the same or another person, creates an `Experiment` value that knows how to interpret the provided `ExperimentData`. It can describe in a very declarative manner what CSV files with the results should be saved, what charts and histograms should be plotted, what statistics summary should be calculated and so on. He/she only has to know those `String` names of the simulation variables (which can be arrays and lists).

Finally, the analyst runs the simulation experiment. It can be either a single simulation, or a parameterized Monte-Carlo simulation, say, consisting of 1000 parallel simulation runs depending on random external parameters.

```
runExperiment :: Experiment -> Simulation ExperimentData -> IO ()
runExperimentParallel :: Experiment -> Simulation ExperimentData -> IO ()
```

The both functions do the same, only the latter launches the simulation runs in parallel[1] as possible. They create an HTML page with the results of the simulation experiment. This page can be observed in the Internet browser.

Returning to our example, the model can be rewritten in the following way:

```
{-# LANGUAGE RecursiveDo #-}
```

---

[1]If the source file was compiled with option `-threaded` and something like `+RTS -N4 -RTS` was passed in to the executable. Please refer to the GHC documentation.

```
...

model :: Simulation ExperimentData
model =
  mdo queue <- newQueue
      a <- integ (- ka * a) 100
      b <- integ (ka * a - kb * b) 0
      c <- integ (kb * b) 0
      let ka = 1
          kb = 1
      experimentDataInStartTime queue
        [("t", seriesEntity "time" time),
         ("a", seriesEntity "a" a),
         ("b", seriesEntity "b" b),
         ("c", seriesEntity "c" c)]
```

Now we can refer to the simulation variables by their `String` names in our experiment that describes what we want to do with our simulation model. Below we want to see the experiment specs, plot the time series and save our results in the CSV file.

```
experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1,
    experimentTitle = "Chemical Reaction",
    experimentDescription = "Chemical Reaction as described in " ++
                            "the 5-minute tutorial of Berkeley-Madonna",
    experimentGenerators =
      [outputView defaultExperimentSpecsView,
       outputView $ defaultTimeSeriesView {
         timeSeriesTitle = "Time Series",
         timeSeries = [Left "a", Left "b", Left "c"] },
       outputView $ defaultTableView {
         tableSeries = ["t", "a", "b", "c"] } ] }
```

Our experiment defines only a single simulation run. Therefore we can use any of the two run functions.

```
main = runExperiment experiment model
```

I saved it in file *ChemicalReactionIntro.hs* and launched from the Terminal application on OS X. In my case I received the following output:

```
bash-3.2$ runghc ChemicalReactionIntro.hs
Using directory experiment
Generated file experiment/Time Series - 1.png
Generated file experiment/Table - 1.csv
Generated file experiment/index.html
```

It says that the new sub-directory *experiment* was generated in my working directory (I could specify any file names in the experiment if I wished) and it contains something that is very similar to the web page. Indeed, the sub-directory contains the main *index.html* file, a chart and a CSV file as shown on figure 2.1.
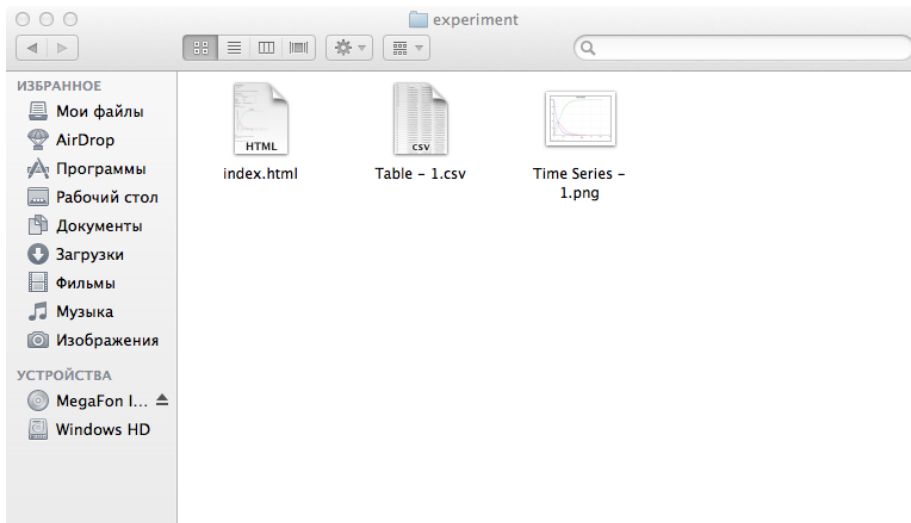
Figure 2.1: The output directory of the simulation experiment.

Opening the generated *index.html* file in the Internet browser, I see a web page with the chart and a hyper-link to the CSV file as shown on figure 2.2.

In my opinion the Aivika experiments are a very powerful tool for automatic saving the results and for providing the preliminary statistical analysis. So, we can apply the parameterized Monte-Carlo simulation and ask Aivika to plot the deviation charts by rule 3-sigma, draw the histograms and collect the summary by the statistics data. Two examples are provided below.

The deviation chart from figure 2.3 was received with help of the next code:

```
outputView $ defaultDeviationChartView {
  deviationChartTitle = "The deviation chart for Net Income and Cash Flow",
  deviationChartSeries = [Left netIncomeName,
                          Left netCashFlowName] },
```

The histogram from figure 2.4 was specified by the following code:

```
outputView $ defaultFinalHistogramView {
  finalHistogramTitle = "Final Histogram - 5",
  finalHistogramPlotTitle = "The distribution of the mean heating time " ++
                            "in the final simulation time point.",
  finalHistogramSeries = [meanHeatingTimeName] },
```

Please refer to the documentation and examples provided with these libraries for more details. Below in this document I will use only models that return relatively simple data and don't use the mentioned two packages. Mastering Aivika depends on the knowledge of the simulation concepts and an ability to use them. So, let us return to the modeling techniques.
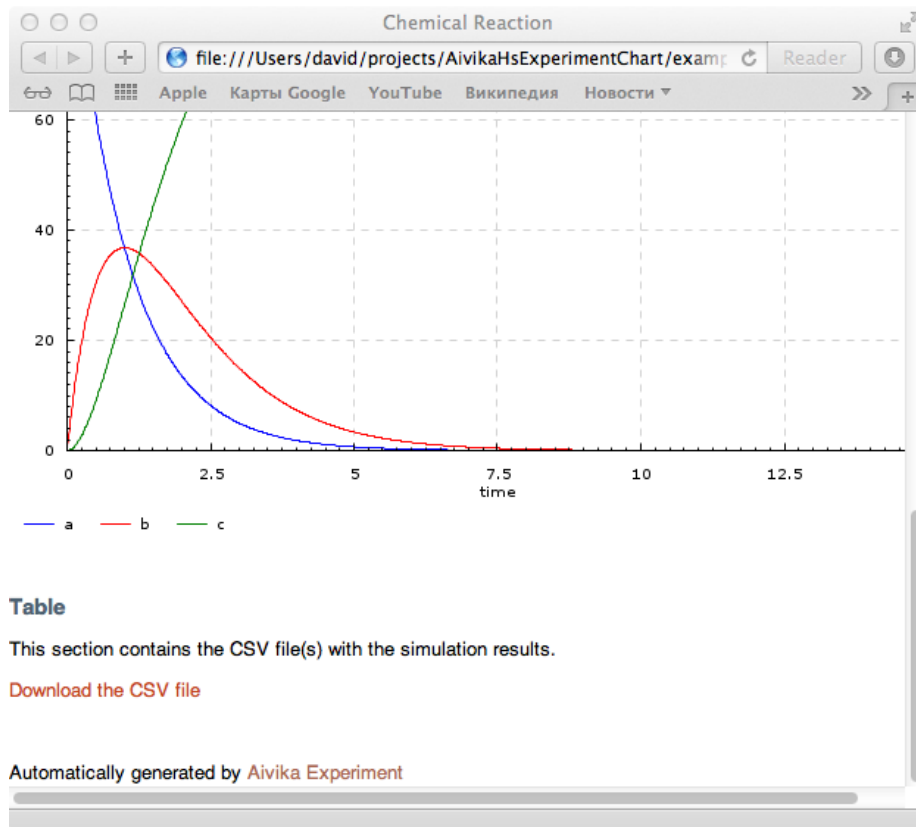
Figure 2.2: Observing the results of the simulation experiment in the Internet browser.

2.3   Summary

Thus, we saw that the system of differential equations can be modeled with help of the `Dynamics` and `Simulation` monads. Actually, the equations can be stochastic as the monads allow embedding any `IO` action such as generating random number values.

In continuation of this subject the next chapter shows how these two monads can be applied to the Discrete Event Simulation.
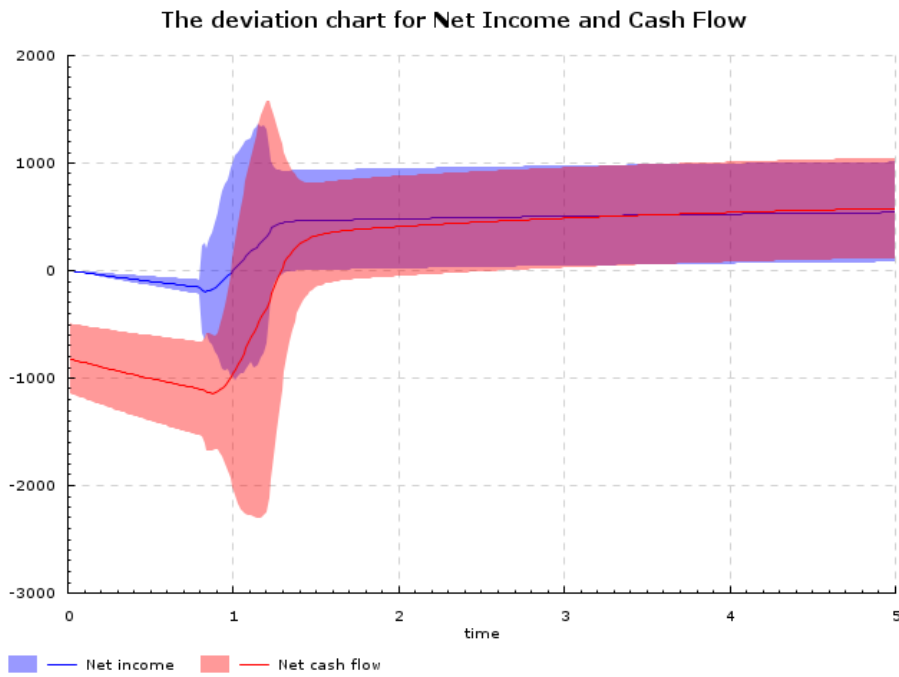
Figure 2.3: The deviation chart by rule 3-sigma for the Financial model as described in Vensim 5 Modeling Guide. The model depends on random external parameters. The Monte-Carlo simulation with 1000 parallel runs was applied.

The distribution of the mean heating time in the final simulation time point.

Figure 2.4: The histogram for the combined discrete-continuous model of the Furnace. The Monte-Carlo simulation with 1000 parallel runs was applied.

Chapter 3

Discrete Event Simulation

The *Discrete Event Simulation (DES)* involves simulating variables that change in discrete steps. Then an event usually implies some variable change. The following three approaches are widely appli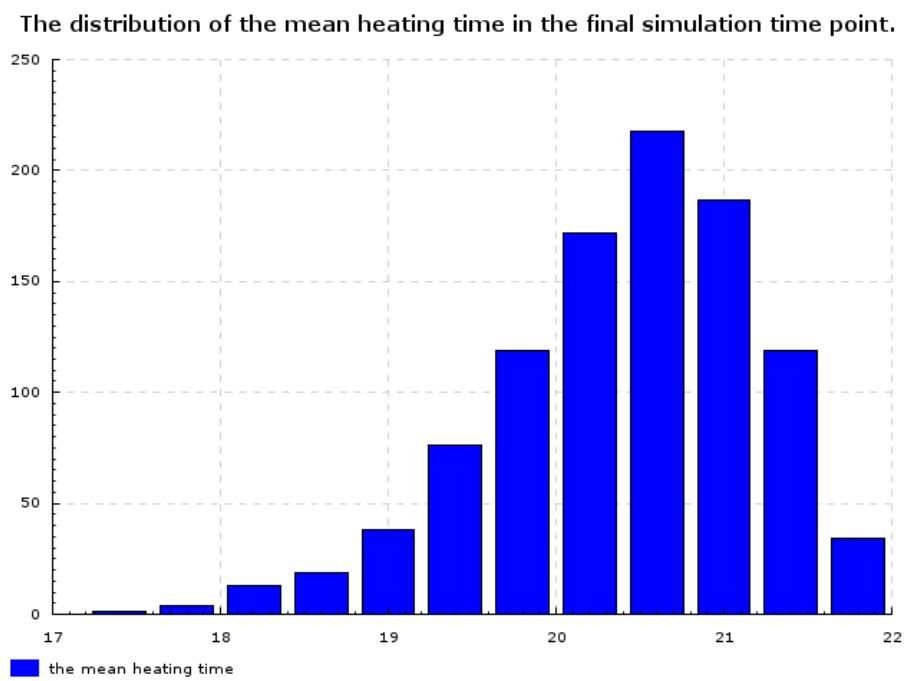ed: activity-oriented, event-oriented and process-oriented. All three are supported by Aivika. In this chapter we will focus on the event-oriented simulation.

Under the *event-oriented* paradigm, 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*.

3.1 Event Queue

An event queue is the heart of the Aivika ecosystem. It coordinates different parts of the model, usually implicitly, and it allows the simulation time to flow synchronously from one entity to another.

There is a plenty of useful functions for working with the event queue. Here and below I will state only the most essential.

```
module Simulation.Aivika.Dynamics.EventQueue

data EventQueue

newQueue :: Simulation EventQueue

enqueue :: EventQueue -> Double -> Dynamics () -> Dynamics ()
enqueue q t m
```

The `newQueue` function creates a new event queue within the `Simulation` computation. The `enqueue` function is rather interesting. It adds the event handler `m` to queue `q`. The event must be raised at time `t`. The result is a `Dynamics` computation. The most exciting thing is that the event handler is also a computation in the `Dynamics` monad. If we want to pass in some message with the event then we should use a closure.

So, the event handler is a dynamic process that has a single purpose to perform some side effect at the specified time.

To functionate properly, the event queue must be involved in the main simulation. Each of the next two functions return a computation that represents a moving force of the queue. They differ only in one thing. The synchronous version doesn't allow calling from the past time point.

```
runQueue :: EventQueue -> Dynamics ()
runQueueSync :: EventQueue -> Dynamics ()
```

You will rarely need to call these functions explicitly in your models as they are usually called implicitly by the built-in simulation entities of Aivika such as references, variables and agents considered further in this document.

It finishes the event queue description. The queue is internally represented as a heap-based priority queue. It is efficiently implemented using imperative algorithms in the `IO` monad.

Before we proceed to an example, I will introduce a reference that can be applied to store, update and read some data within the simulation.

3.2   References and Variables

A value of the `Ref` type is like the standard `IORef` value except for one thing. The former is bound to the event queue. Before the reference value is requested in some time point, the corresponded queue is checked whether there are pending events that should be raised. It makes the model coordinated.

```
module Simulation.Aivika.Dynamics.Ref

data Ref a

newRef :: EventQueue -> a -> Simulation (Ref a)

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

Using the references, different parts of the model can communicate to each other. If these references are bound to the same event queue then this communication will be coordinated. In general, this is a good rule to define only one event queue for the entire model.

Actually, the common event queue could be embedded in the `Simulation` computation, which would simplify the API interface of the creation functions. But I left an opportunity for defining several event queues that might be run in parallel within the same simulation run. Then every event queue would have its own execution thread. But this is not implemented yet.

Meantime, if you need an analog of the `Ref` reference but that would keep the history of past values, which can be important for combining the differential equations with the DES model, then you should look at the `Var` type and its

unboxed version `UVar`. They have a similar interface and plus some functions for extracting the history of values. Such `UVar` variables can be safely used within the differential equations.

Each time the reference or variable changes, it emits a new `Signal`. Using namely this property, the Aivika Experiment package[8] is notified about every change of the observed simulation variables. But this is a subject of separate description.

3.3 Example MachRep1

Now it is time to illustrate the simulation approach. I will use the following task [1].

> 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.

We create the `Simulation` computation that returns the long-run proportion. Here we need an auxiliary function to generate exponentially distributed random values. Then we run the simulation using the specified specs.

```
import System.Random
import Control.Monad.Trans

import Simulation.Aivika.Dynamics
import Simulation.Aivika.Dynamics.Simulation
import Simulation.Aivika.Dynamics.Base
import Simulation.Aivika.Dynamics.EventQueue
import Simulation.Aivika.Dynamics.Ref

upRate = 1.0 / 1.0        -- reciprocal of mean up time
repairRate = 1.0 / 0.5    -- reciprocal of mean repair time

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

exprnd :: Double -> IO Double
exprnd lambda =
  do x <- getStdRandom random
     return (- log x / lambda)

model :: Simulation Double
model =
  do queue <- newQueue
     totalUpTime <- newRef queue 0.0

     let machineBroken :: Double -> Dynamics ()
```

```
            machineBroken startUpTime =

               do finishUpTime <- time
                   modifyRef totalUpTime (+ (finishUpTime - startUpTime))
                   repairTime <- liftIO $ exprnd repairRate

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

            machineRepaired :: Dynamics ()
            machineRepaired =

               do startUpTime <- time
                   upTime <- liftIO $ exprnd upRate

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

        runDynamicsInStartTime $
          do t0 <- starttime
              -- start the first machine
              enqueue queue t0 machineRepaired
              -- start the second machine
              enqueue queue t0 machineRepaired

        runDynamicsInStopTime $
          do x <- readRef totalUpTime
              y <- stoptime
              return $ x / (2 * y)

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

Parameter spcDT of the simulation specs is not actually used here by Aivika. The event queue doesn't rely on the integration time points. It has its own order of calculations concerning only with those time points at which the events must be processed. The event queue is involved in the simulation through the runQueueSync function, which is called implicitly when we call the readRef function.

Here the event handlers are created by function machineBroken and value machineRepaired. The latter is just a computation that has type Dynamics (). The former is a function that accepts one argument. Given the start up time, this function creates a computation of type Dynamics () too. In such a way we can transfer with the event any data we want.

```
-- start the first machine
enqueue queue t0 machineRepaired

-- start the second machine
enqueue queue t0 machineRepaired
```

In this block we initialize the event queue passing in two events which should be raised at the initial time of simulation. Each of the events corresponds to a

separate machine. We begin with the state at which the machine is repaired. Note that the block is launched in the start time point within the `Simulation` computation.

To switch from the repaired state to the broken one, we calculate the time at which the machine should be broken and create a new event passing in the start up time in the closure.

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

After the machine is broken it must be repaired during the random time with the specified rate. After this time is over the machine becomes repaired, about which we add the corresponded event to the queue.

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

During the repair time we update our counter using the `modifyRef` function.

```
modifyRef totalUpTime (+ (finishUpTime - startUpTime))
```

Finally, we request for the value in the last integration time point using the `runDynamicsInStopTime` function. It calls the `readRef` function that in its turn unwinds all the events starting from the initial integration time point, because the reference is bound up with the event queue.

The next chapter shows how the same model can be simulated using the process-oriented approach.

Chapter 4

Process-oriented Simulation

Under the *process-oriented* paradigm, 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.

4.1 Discontinuous Processes

Aivika provides a special kind of processes which I will call *discontinuous processes* to distinguish them from the defined earlier dynamic processes that had type `Dynamics`. The discontinuous processes are important for the process-oriented simulation. These processes can suspend at any time and then resume later. It allows us to model more easily some activities.

So, a discontinuous process is a value of polymorphic type `Process`. In most cases it can behave like the dynamic process. Indeed, any `Dynamics` computation can be lifted to the `Process` computation with help of the `liftDynamics` function. For example, expression `liftDynamics time` returns the current simulation time as the `Process` computation.

The `Process` type is a monad. It allows us to build discontinuous processes from small building pieces.

```
instance Functor Process
instance Monad Process
instance MonadIO Process
```

The main characteristic of the discontinuous process is its ability to suspend. The next function suspend the current `Process` computation for the specified time.

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

The process can also be *passivated*. Somewhere it is like a suspension but lasts for an indefinite period of time. The current process is stopped and waits for a moment until somebody else *reactivates* it.

```
passivateProcess :: Process ()
```

The difference between the hold and passivation is that the hold process stops and adds an awakening event to the event queue that acts behind the scene. Such a process is resumed right after the corresponded event is raised. On the contrary, the passivated process stops and stores its continuation in a special structure called a *process ID*.

The process ID is actually a handle. Each process is bound up with its handle. They are one. We can use only unique handles. Two handles can be tested for equality. Also we can request the process for its handle.

```
module Simulation.Aivika.Dynamics.Process

data ProcessID
instance Eq ProcessID

processID :: Process ProcessID
```

To reactivate another process, we must know its ID. Also we can test whether a process with the specified ID is passivated.

The next two functions don't affect the current computation. The reactivated process is always started in another computation. Therefore the returned below values are the `Dynamics` computations. They cannot stop the current process, being called from it regardless of whether it is the `Dynamics` or `Process` computation.

```
reactivateProcess :: ProcessID -> Dynamics ()
processPassive :: ProcessID -> Dynamics Bool
```

A time of creating the process ID and a time of starting the discontinuous process are separated. It allows us to create IDs, define some logic for the processes that would use these IDs and then already launch the processes.

```
newProcessID :: EventQueue -> Simulation ProcessID
runProcess :: Process () -> ProcessID -> Double -> Dynamics ()
```

The `newProcessID` function requires an event queue and creates an unique process ID. The queue acts behind the scene each time we hold the process for the specified time or reactivate it.

The `runProcess` function runs the discontinuous process at the specified time. We must assign an unique ID to the new process.

The next section shows how we can apply the discontinuous processes to the simulation.

4.2   Revised Example MachRep1

Now I will show how the model[1] from section 3.3 can be rewritten using the discontinuous processes.

For simplicity, the problem statement is repeated.

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 main idea is to represent every machine as a separate discontinuous process, i.e. a computation in the Process monad.

```
import System.Random
import Control.Monad.Trans

import Simulation.Aivika.Dynamics
import Simulation.Aivika.Dynamics.Base
import Simulation.Aivika.Dynamics.Simulation
import Simulation.Aivika.Dynamics.EventQueue
import Simulation.Aivika.Dynamics.Ref
import Simulation.Aivika.Dynamics.Process

upRate = 1.0 / 1.0        -- reciprocal of mean up time
repairRate = 1.0 / 0.5   -- reciprocal of mean repair time

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

exprnd :: Double -> IO Double
exprnd lambda =
  do x <- getStdRandom random
     return (- log x / lambda)

model :: Simulation Double
model =
  do queue <- newQueue
     totalUpTime <- newRef queue 0.0

     pid1 <- newProcessID queue
     pid2 <- newProcessID queue

     let machine :: Process ()
         machine =
           do startUpTime <- liftDynamics time
              upTime <- liftIO $ exprnd upRate
              holdProcess upTime
              finishUpTime <- liftDynamics time
              liftDynamics $
                modifyRef totalUpTime
                (+ (finishUpTime - startUpTime))
              repairTime <- liftIO $ exprnd repairRate
              holdProcess repairTime
              machine

     runDynamicsInStartTime $
       do t0 <- starttime
```

```
        runProcess machine pid1 t0
        runProcess machine pid2 t0

   runDynamicsInStopTime $
     do x <- readRef totalUpTime
        y <- stoptime
        return $ x / (2 * y)

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

As before, the integration time step `spcDT` has no any sense for this model but we have to define it, though. In case of the hybrid model the `spcDT` parameter would play already an important role. But here the discontinuous processes are implemented on top of the event queue that doesn't use `spcDT`.

What is new is that how the machine is constructed. It is defined as a discontinuous process that looks like an infinite loop which is terminated automatically after the simulation is complete. In this loop we model the work of the machine.

To get the current simulation time, we use the `time` built-in that returns a computation of type `Dynamics Double`. Such a computation must be lifted to be involved in the upper `Process` computation which has another type. Therefore we apply the `liftDynamics` function. In such a way we can receive the current simulation time within the `Process` computation.

```
startUpTime <- liftDynamics time
```

In the same way we can receive the current value of any computation in the `Dynamics` monad, including the integrals. It allows us to truly build hybrid models.

After we receive the current simulation time and calculate the up time, we suspend the current process.

```
holdProcess upTime
```

In the specified time the process will resume and its control flow will continue. Then we update the counter, calculate the repair time and suspend the process again.

After the process resumes at the second time we repeat all calling the process computation recursively. It is safe enough from the technical point of view if you are interested in details but this subject goes beyond this document. In short, it won't consume the stack.

To initiate two separate processes at the start time of simulation, we use the `runProcess` function, which is called in the initial time point.

```
runProcess machine pid1 t0
runProcess machine pid2 t0
```

Note that the process IDs must be different. It will be a run-time error if the already used ID is used again.

Before we proceed to more complex models I will describe that how in Aivika we can model the limited resources.

4.3 Resources

In Aivika the limited resources are modeled with help of the `Resource` data type. We pass in an event queue and the initial count to the `newResource` function that creates a new resource within the `Simulation` computation.

```
module Simulation.Aivika.Dynamics.Resource

data Resource
instance Eq Resource

newResource :: EventQueue -> Int -> Simulation Resource
newResourceWithCount :: EventQueue -> Int -> Int -> Simulation Resource
```

The event queue is needed to suspend those discontinuous processes that try to request for the resource in case of its deficiency. In general, to acquire the next unit of the resource, we call the `requestResource` function within the `Process` computation.

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

If the resource is available then its count is decreased; otherwise, the process is suspended until some other process releases the previously acquired resource with help of the next function.

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

Any acquired resource must be released. It will be a logical error if you release the resource that was not acquired with help of the `requestResource` function. It would be too costly to track such errors. Therefore this is your responsibility to release the acquired resources.

To receive the available count of the limited resource, we can call function `resourceCount`. The second function returns immediately the initial count of the specified resource. The third one returns the event queue that actually manages the resource and processes behind the scene.

```
resourceCount :: Resource -> Dynamics Int
resourceInitCount :: Resource -> Int
resourceQueue :: Resource -> EventQueue
```

This small set of new functions allows us to build models with more complex behavior. Please refer to the documentation which contains more functions for working with the resources.

4.4 Example MachRep2

Let us go on with the following task[1].

Two machines, but sometimes break down. Up time is exponen-
tially distributed with mean 1.0, and repair time is exponentially
distributed with mean 0.5. In this example, there is only one re-
pairperson, 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.

Now we have to work with the limited resource, namely the repairperson.
In many places the model is similar to the previous one. Only the block in
which the machines are repaired are guarded by functions `requestResource`
and `releaseResource`. Also we add two new counters.

```
import System.Random
import Control.Monad
import Control.Monad.Trans

import Simulation.Aivika.Dynamics
import Simulation.Aivika.Dynamics.Simulation
import Simulation.Aivika.Dynamics.Base
import Simulation.Aivika.Dynamics.EventQueue
import Simulation.Aivika.Dynamics.Ref
import Simulation.Aivika.Dynamics.Resource
import Simulation.Aivika.Dynamics.Process

upRate = 1.0 / 1.0        -- reciprocal of mean up time
repairRate = 1.0 / 0.5   -- reciprocal of mean repair time

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

exprnd :: Double -> IO Double
exprnd lambda =
  do x <- getStdRandom random
     return (- log x / lambda)

model :: Simulation (Double, Double)
model =
  do queue <- newQueue

     -- number of times the machines have broken down
     nRep <- newRef queue 0

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

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

```
    repairPerson <- newResource queue 1

    pid1 <- newProcessID queue
    pid2 <- newProcessID queue

    let machine :: Process ()
        machine =
          do startUpTime <- liftDynamics time
             upTime <- liftIO $ exprnd upRate
             holdProcess upTime
             finishUpTime <- liftDynamics time
             liftDynamics $ modifyRef totalUpTime
               (+ (finishUpTime - startUpTime))

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

             requestResource repairPerson
             repairTime <- liftIO $ exprnd repairRate
             holdProcess repairTime
             releaseResource repairPerson

             machine

    runDynamicsInStartTime $
      do t0 <- starttime
         runProcess machine pid1 t0
         runProcess machine pid2 t0

    runDynamicsInStopTime $
      do x <- readRef totalUpTime
         y <- stoptime
         n <- readRef nRep
         nImmed <- readRef nImmedRep
         return (x / (2 * y),
                 fromIntegral nImmed / fromIntegral n)

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

We create two new counters to find the proportion of the time that a given machine does not have immediate access to the repairperson.

```
nRep <- newRef queue 0
nImmedRep <- newRef queue 0
```

Also there is only one repairperson. The corresponded resource is created in the following line:

```
repairPerson <- newResource queue 1
```

To check whether the repairperson is free or busy, we use the `resourceCount` function. The next code increases the second counter only if he/she is free. If the repairperson is busy then **n** equals 0.

```
liftDynamics $
 do modifyRef nRep (+ 1)
    n <- resourceCount repairPerson
    when (n == 1) $
      modifyRef nImmedRep (+ 1)
```

To repair the broken machine, we have to acquire the resource busying the repairperson. This operation suspends the current discontinuous process if he/she is already busy with another machine.

```
requestResource repairPerson
```

After the resource is acquired, the repairing process is modeled as a short-time suspension of the current process. Then the machine is counted repaired and we must release the resource, i.e. free the repairperson.

```
releaseResource repairPerson
```

Then we repeat the loop recursively calling the same computation. It should be a general rule in modeling the discontinuous processes.

The next example is more complicated and involves a process passivation and the following reactivation.

4.5   Example MachRep3

The next model[1] has a more complex behavior.

> Variation of the previous models. 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, and she is not summoned until both machines are down. We find the proportion of up time. It should come out to about 0.45.

To model the work of two machines, we have to passivate the first broken machine until the second machine is broken too. Then we summon the repairperson, reactivating the first machine. Therefore the discontinuous process that models the machine must know the process ID of another machine. We pass in it as an argument.

```
import System.Random
import Control.Monad
import Control.Monad.Trans

import Simulation.Aivika.Dynamics
import Simulation.Aivika.Dynamics.Simulation
import Simulation.Aivika.Dynamics.Base
import Simulation.Aivika.Dynamics.EventQueue
import Simulation.Aivika.Dynamics.Ref
import Simulation.Aivika.Dynamics.Resource
```

```
import Simulation.Aivika.Dynamics.Process

upRate = 1.0 / 1.0        -- reciprocal of mean up time
repairRate = 1.0 / 0.5   -- reciprocal of mean repair time

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

exprnd :: Double -> IO Double
exprnd lambda =
  do x <- getStdRandom random
     return (- log x / lambda)

model :: Simulation Double
model =
  do queue <- newQueue

     -- number of machines currently up
     nUp <- newRef queue 2

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

     repairPerson <- newResource queue 1

     pid1 <- newProcessID queue
     pid2 <- newProcessID queue

     let machine :: ProcessID -> Process ()
         machine pid =
           do startUpTime <- liftDynamics time
              upTime <- liftIO $ exprnd upRate
              holdProcess upTime
              finishUpTime <- liftDynamics time
              liftDynamics $ modifyRef totalUpTime
                (+ (finishUpTime - startUpTime))

              liftDynamics $ modifyRef nUp $ \a -> a - 1
              nUp' <- liftDynamics $ readRef nUp
              if nUp' == 1
                then passivateProcess
                else liftDynamics $
                       do n <- resourceCount repairPerson
                          when (n == 1) $
                            reactivateProcess pid

              requestResource repairPerson
              repairTime <- liftIO $ exprnd repairRate
              holdProcess repairTime
              liftDynamics $ modifyRef nUp $ \a -> a + 1
              releaseResource repairPerson

              machine pid

     runDynamicsInStartTime $
```

```
        do t0 <- starttime
           runProcess (machine pid2) pid1 t0
           runProcess (machine pid1) pid2 t0

     runDynamicsInStopTime $
       do x <- readRef totalUpTime
          y <- stoptime
          return $ x / (2 * y)

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

After the machine is broken, we decrease the counter of machines currently up. If only this machine is broken then we passivate it. Otherwise, the both machines are counted broken and the last of them, i.e. current, reactivates another in that case if the repairperson is free, i.e. **n** equals 1.

```
liftDynamics $ modifyRef nUp $ \a -> a - 1
nUp' <- liftDynamics $ readRef nUp
if nUp' == 1
  then passivateProcess
  else liftDynamics $
       do n <- resourceCount repairPerson
          when (n == 1) $
            reactivateProcess pid
```

To repair the machine, we acquire the resource. Before we release it, we increase the counter of the machines.

```
liftDynamics $ modifyRef nUp $ \a -> a + 1
releaseResource repairPerson
```

Each of the both machines must know of another. We pass in other's ID during the start of the machine.

```
runProcess (machine pid2) pid1 t0
runProcess (machine pid1) pid2 t0
```

The rest part should be familiar.

This model can be trivially converted[1] to the Aivika experiment[8, 9] and then analyzed, for example, what distribution the output variable has in the final simulation time point. The result of such an analysis for parallel 10000 simulation runs in form of the histogram is shown on figure 4.1. The simulation with generating this histogram and other charts, collecting the statistics summary and so on, lasted only for 1m28.914s on my MacBook Pro.

In Aivika the process-oriented simulation is built on top of the event-driven simulation. The next chapter shows how the activity-oriented simulation can also be built on top of the event-driven to be included safely in the main simulation.

---

[1]We have to return `Simulation ExperimentData` from the model, define how to interpret these data in an `Experiment` value and, finally, run the experiment.
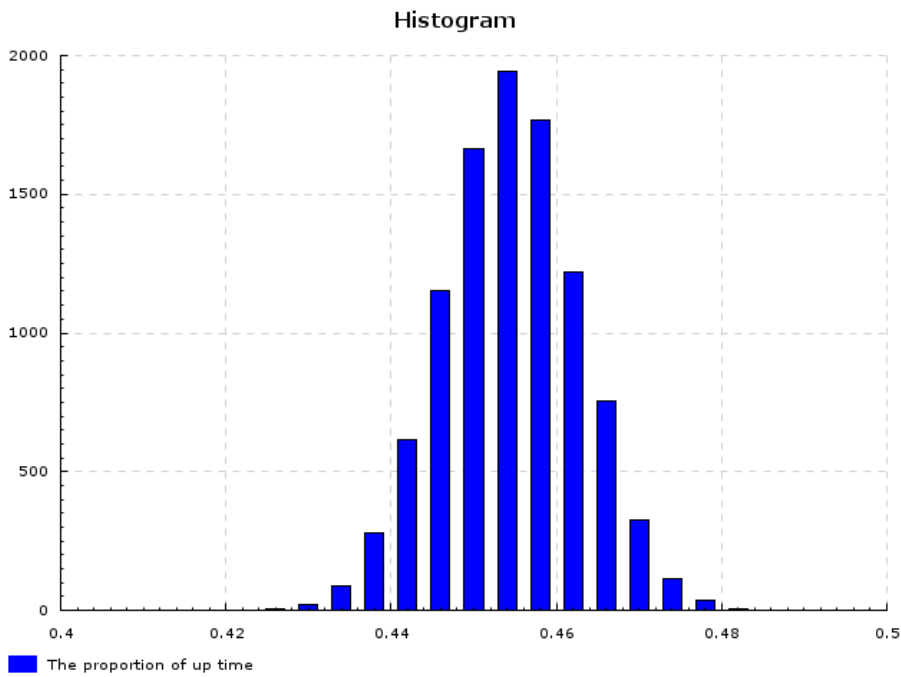
Figure 4.1: The distribution of the output variable for task MachRep3. The Monte-Carlo simulation with parallel 10000 runs was applied.

Chapter 5

Activity-oriented Simulation

Under the *Activity-oriented* paradigm, 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*.

In Aivika we have the time built-ins. The `dt` computation can play a role of the tiny time increment, although you can specify any other time intervals and they can be irregular. Also the `Dynamics` type is a monad. Therefore we can define a rather complex code in the monad computation including that one which is necessary to operate on activities. It would be tempting to use this in the models.

Then in the Haskell programming language the activity-oriented model will just be a computation of type `Dynamics ()`. To include this computation in the main simulation, we can use the event queue and one of its functions like these:

```
enqueueWithTimes :: EventQueue -> [Double] -> Dynamics () -> Dynamics ()
enqueueWithIntegTimes :: EventQueue -> Dynamics () -> Dynamics ()
```

The first function actuates the event handler in the specified time points. The second one actuates sequentially the event handlers precisely in the integration time points.

The functions allow us to safely involve the activity-oriented model in the main simulation. More generally, if you want to perform some actions sequentially in the specified time points then you should use one of these two functions. They guarantee that these actions will be sequential and that these actions will be synchronized with the event queue.

Moreover, you can generate some data in the activity-oriented model, for example, reading some information from the file and processing it. To pass in these data to the main model, you can either use the `Ref` references and `Var` variables (that have a memory of past values), or trigger signals with help of the `Signal` type. The latter is a generalization of the concept of the .NET event,

but oriented for working with the dynamic system. This subject goes beyond the scope of this introductory document.

Now it is time of some practice.

5.1   Example MachRep1 Again

I will take the model[1] from section 3.3. For easiness, I will give the model description again.

> 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.

We have much manual work to do. We have to track each iteration. We create two counters of iterations. The first counter defines how long the machine is in a working state. The second counter defines how long the machine is broken.

The code is quite routine. The most interesting thing is that how we involve the activity-oriented model in the main simulation. Note that this model is less accurate than the previous ones as it rounds the time intervals[1]. Also it can be more slow. Such models will be usually more slow as we have to take a relatively tiny time increment to model the problem adequately.

```
import System.Random
import Control.Monad.Trans

import Simulation.Aivika.Dynamics
import Simulation.Aivika.Dynamics.Simulation
import Simulation.Aivika.Dynamics.Base
import Simulation.Aivika.Dynamics.EventQueue
import Simulation.Aivika.Dynamics.Ref

upRate = 1.0 / 1.0       -- reciprocal of mean up time
repairRate = 1.0 / 0.5   -- reciprocal of mean repair time

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

exprnd :: Double -> IO Double
exprnd lambda =
  do x <- getStdRandom random
     return (- log x / lambda)
```

---

[1]Such an inaccuracy is not necessary as we could use exact time intervals but it would probably require to use the recursive *do*-notation as we would have to calculate these intervals **after** they would be passed in to some variation of the **enqueue** function which we would probably have to write manually. But this is an open question.

```
model :: Simulation Double
model =
  do queue <- newQueue
     totalUpTime <- newRef queue 0.0

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

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

              -- a number of iterations when
              -- the machine is broken
              repairNum <- newRef queue (-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' <- time
                            dt' <- dt
                            modifyRef totalUpTime $
                              \a -> a +
                              (finishUpTime' - startUpTime')
                            repairTime' <-
                              liftIO $ exprnd repairRate
                            writeRef repairNum $
                              round (repairTime' / dt')

                       repaired =
                         do writeRef repairNum (-1)
                            -- the machine is repaired
                            t'  <- time
                            dt' <- dt
                            writeRef startUpTime t'
                            upTime' <-
                              liftIO $ exprnd upRate
                            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 Dynamics ()
    m1 <- machine
    m2 <- machine

    -- start the time-driven simulation of the machines
    -- through the event queue
    runDynamicsInStartTime $
      do enqueueWithIntegTimes queue m1
         enqueueWithIntegTimes queue m2

    -- return the result in the stop time
    runDynamicsInStopTime $
      do x <- readRef totalUpTime
         y <- stoptime
         return $ x / (2 * y)

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

The activity-oriented simulation requires much manual work. Compare with
that how easily we could define the same model under the event-oriented and
process-oriented paradigms.

As before, the activity-oriented simulation uses the event queue. The next
chapter shows how the same queue can be applied to modeling the agents.

Chapter 6

Agent-based Modeling

The agent-based modeling is quite different in comparison with DES and System Dynamics. The main entity is an *agent* that acts as a *state machine*. The states can have children. The states can be activated, or deactivated. All ancestors of the active state are considered implicitly active, but there is always only one selected active state.

The state hierarchy represents a *forest of trees*. We can modify this forest dynamically during simulation. We can add new states, define their activation and deactivation computations and then make some of these states active, selecting one of them as the down-most active state. Its ancestor line will be activated. Other states will be deactivated if required. The ancestor remains activated when the new selected state also belongs to its line. The states are activated and deactivated only in case of need.

Also we can assign the timer and timeout handlers to each active state. These handlers are computations that are actuated in the specified amount of time. This is what gives a moving force to the agents making them an excellent tool for modeling some systems.

Aivika supports the basic constructs for the agent-based modeling. As before, this support is based on the simulation monads. The activation and deactivation procedures are the `Dynamics` computations. So are the timer and timeout handlers. Everything is managed by the event queue.

6.1   Agents

Every agent is bound up with the specified event queue. Every state is bound up with its agent. Also any state can have a parent state.

```
module Simulation.Aivika.Dynamics.Agent

data Agent
data AgentState

instance Eq Agent
```

```
instance Eq AgentState

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

agentQueue :: Agent -> EventQueue
stateAgent :: AgentState -> Agent
stateParent :: AgentState -> Maybe AgentState
```

The agent may have a selected active state. If it is defined then it is always the down-most state in the line of active states. All ancestors of this state in the hierarchy forest are considered implicitly active. Other states are deactivated. To request for this down-most active state, we can apply the `agentState` function.

```
agentState :: Agent -> Dynamics (Maybe AgentState)
```

If the agent was not initiated yet then it has no active state and this function returns `Nothing` within the `Dynamics` computation.

We can initiate the agent and select a new down-most active state with help of the same function. This is function `activateState`.

```
activateState :: AgentState -> Dynamics ()
```

Each state has the activation and deactivation computations. They are actuated if necessary. By default they do nothing. We can modify them with help of the following two functions.

```
setStateActivation :: AgentState -> Dynamics () -> Simulation ()
setStateDeactivation :: AgentState -> Dynamics () -> Simulation ()
```

Sometimes, we need to set the transition state for the current state. The former will be activated right after the latter is activated. It allows activating the nested states sequentially.

```
setStateTransition :: AgentState -> Dynamics (Maybe AgentState) -> Simulation ()
```

What makes the agents interesting for the modeling is the timeout and timer handlers. These handlers are similar to the event handlers we saw before and they indeed use the event queue under the hood. Only the timeout and timer handlers are assigned to some state and they are legitimate while the corresponded state remains active. After the state is deactivated all its handlers become outdated and then they are ignored. But you can assign again new handlers at time of next activating the state.

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

The first argument is the state which the handler is assigned to. The second argument specifies the time period in which the handler can be actuated, if the state will remain active. The third argument defines the corresponded computation.

If the timeout handler is ever actuated then it finishes after the work. On the contrary, the timer handler adds itself again. It will periodically repeat while the state remains active. Therefore the time period is defined as a pure value in the first case and as a computation in the second case. If the time period is defined as a number then it remains always constant. But if the time period is defined as the `Dynamics` computation then it is recalculated each time the timer handler adds itself again.

The event queue manages all the process here. Aivika creates wrappers for the timer and timeout handlers and puts these wrappers in the event queue. Each agent state has an internal version number. When Aivika creates the handler's wrapper, it captures the current version value of the state in the closure. If the state becomes later deactivated then its internal version increases, which makes the handler's wrapper with less version number outdated. In such a case the timeout or timer handler is not actuated even if its wrapper is still actuated by the event queue. This approach is very simple and efficient enough.

Now we will see how this theory can be applied to practice.

6.2   Example BassDiffusion

An agent-based version of the Bass Diffusion model[2] is described in the Any-Logic tutorial.

> 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.

The model begins similarly. We import the modules as well as define constants, simulation specs and two random functions.

```
import System.Random
import Data.Array
import Control.Monad
import Control.Monad.Trans

import Simulation.Aivika.Dynamics
import Simulation.Aivika.Dynamics.Simulation
import Simulation.Aivika.Dynamics.EventQueue
import Simulation.Aivika.Dynamics.Agent
```

```
import Simulation.Aivika.Dynamics.Ref

n = 500     -- the number of agents

advertisingEffectiveness = 0.011
contactRate = 100.0
adoptionFraction = 0.015

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

exprnd :: Double -> IO Double
exprnd lambda =
  do x <- getStdRandom random
     return (- log x / lambda)

boolrnd :: Double -> IO Bool
boolrnd p =
  do x <- getStdRandom random
     return (x <= p)
```

Now we create an agent identified with the person who can be in two states: an adopter or potential adopter. To create the person, we need the event queue. We place all persons in the array. We need this array to have an access to random agents at time when the specified adopter tries to convert somebody to an adopter too.

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

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

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

The agents and states are created within the `Simulation` computation. Although we could use the recursive *do*-notation to define in one block, here we separate two different steps. At the first step we create the objects. At the second step we define their activation and deactivation computations.

```
definePerson :: Person -> Array Int Person -> Ref Int -> Ref Int -> Simulation ()
```

```
definePerson p ps potentialAdopters adopters =
  do setStateActivation (personPotentialAdopter p) $
       do modifyRef potentialAdopters $ \a -> a + 1
          -- add a timeout
          t <- liftIO $ exprnd advertisingEffectiveness
          let st  = personPotentialAdopter p
              st' = personAdopter p
          addTimeout st t $ activateState st'
     setStateActivation (personAdopter p) $
       do modifyRef adopters  $ \a -> a + 1
          -- add a timer that works while the state is active
          let t = liftIO $ exprnd contactRate    -- many times!
          addTimer (personAdopter p) t $
            do i <- liftIO $ getStdRandom $ randomR (1, n)
               let p' = ps ! i
               st <- agentState (personAgent p')
               when (st == Just (personPotentialAdopter p')) $
                 do b <- liftIO $ boolrnd adoptionFraction
                    when b $ activateState (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 -> Simulation ()
definePersons ps potentialAdopters adopters =
  forM_ (elems ps) $ \p ->
  definePerson p ps potentialAdopters adopters
```

When the potential adopter state is activated, we add a timeout handler with the specified period after which the agent becomes an adopter. The most difficult part is the activation computation for the adopter state. We add a timer handler that periodically calls a procedure, where the adopter is trying to make a random agent an adopter too. Note that the time period for the timer is specified as the `Dynamics` computation (through `liftIO`). It will be recalculated at every next call giving different random numbers.

```
activatePerson :: Person -> Dynamics ()
activatePerson p = activateState (personPotentialAdopter p)

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

The agent activation is straightforward enough. Every agent begins with the potential adopter state.

```
model :: Simulation [IO [Int]]
model =
  do q <- newQueue
     potentialAdopters <- newRef q 0
     adopters <- newRef q 0
     ps <- createPersons q
     definePersons ps potentialAdopters adopters
     runDynamicsInStartTime $
```

```
      activatePersons ps
    runDynamicsInIntegTimes $
      do i1 <- readRef potentialAdopters
         i2 <- readRef adopters
         return [i1, i2]

main =
  do xs <- runSimulation model specs
     forM_ xs $ \x -> x >>= print
```

The remained part is simple. We create agents, define and then activate them. We return the values defined with help of the references. These references are updated by the agents during their work.

Here is one of the possible results of simulation:

```
[[500,0],[499,1],[498,2],[498,2],[498,2],[498,2],[495,5],[495,5],
 [494,6],[488,12],[488,12],[484,16],[480,20],[478,22],[474,26],
 [469,31],[458,42],[448,52],[441,59],[434,66],[426,74],[413,87],
 [403,97],[389,111],[375,125],[363,137],[348,152],[336,164],
 [323,177],[299,201],[281,219],[255,245],[239,261],[216,284],
 [202,298],[187,313],[170,330],[156,344],[141,359],[123,377],
 [114,386],[99,401],[83,417],[78,422],[68,432],[61,439],[56,444],
 [51,449],[46,454],[42,458],[33,467],[30,470],[28,472],[25,475],
 [23,477],[22,478],[20,480],[18,482],[17,483],[11,489],[8,492],
 [7,493],[7,493],[7,493],[5,495],[4,496],[3,497],[3,497],[3,497],
 [2,498],[2,498],[2,498],[2,498],[2,498],[2,498],[2,498],[2,498],
 [2,498],[2,498],[2,498],[2,498]]
```

This model can be easily converted[1] to the Aivika experiment[8, 9] and then analyzed how sensitive is the model. The result of such an analysis with parallel 1000 simulation runs in form of the deviation chart by rule 3-sigma is shown on figure 6.1. On my MacBook Pro the Monte-Carlo simulation with generation of the chart lasted for 9m59.599s[2].

The next chapter returns us to the system of differential equations which we started with.

---

[1]We have to return `Simulation ExperimentData` from the model, define how to interpret these data in an `Experiment` value and, finally, run the experiment.

[2]I used options `+RTS -N4 -RTS` on my dual-core Intel processor with support of hyper-threading after compiled the source file with option `-threaded`.
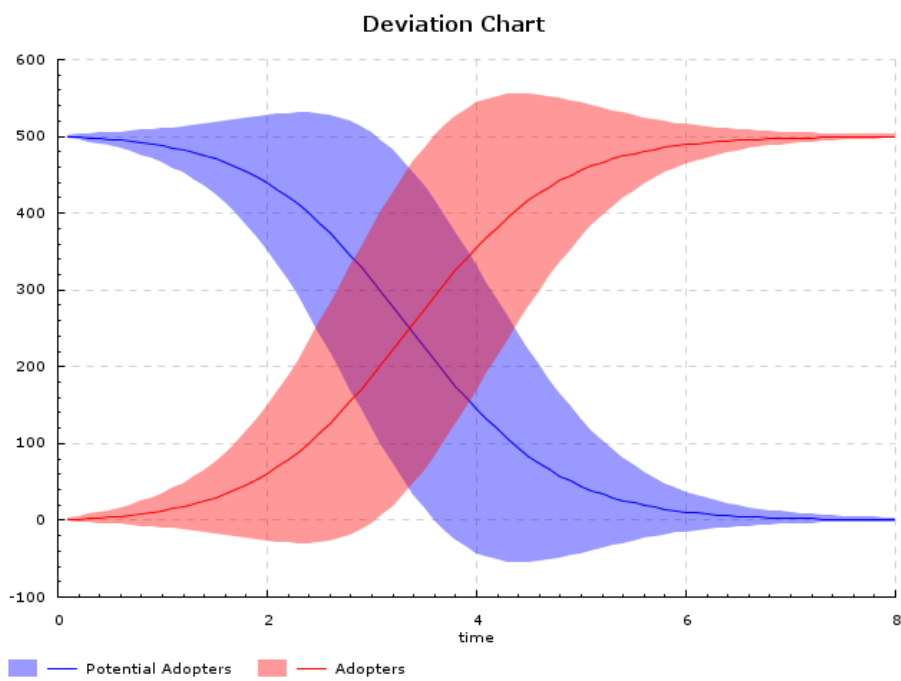
Figure 6.1: It shows how sensitive is the Bass Diffusion model with 500 agents. The Monte-Carlo simulation with parallel 1000 runs was applied.

Chapter 7

System Dynamics

The System Dynamics approach allows us to create dynamic system with loop-backs. Usually, the model is represented as a system of differential equations (ODEs). It can have *stocks* such as *reservoirs*, can have *flows* and *auxiliaries*. The reservoir is just an integral. Then the flow is a summand of the derivative. We take it with the plus or minus sign depending on that whether the flow is inflow or outflow. The auxiliaries correspond to other variables.

There are also discrete stocks such as *conveyors*, *ovens* and *queues*. It is important that their simulation can also be described in terms of the integrating technique such as the Runge-Kutta method or Euler's method. Any stock has a state varying in time. We update sequentially this state in all integration time points. It looks like that as we would integrate numerically differential equations.

An idea of the System Dynamics discipline is to define the model both graphically on the diagram and in the equations. The loopbacks are usually explicitly shown on the diagram which is called a *Stock and Flow Map*. Stocks are an origin of these loopbacks. Flows close them.

The ordinary differential equations and difference equations can be defined in Aivika using the same simulation monads considered above and applying the recursive *do*-notation that allows creating recursive bindings for the simulation variables.

Before we proceed to an example, I will introduce the table functions that are very useful in such equations. Please refer to the documentation, where you can find a plenty of other functions including the delay functions of different kind and order.

Also you can find a rather sophisticated example of the financial model as described in Vensim 5 Modeling Guide in the distributive of Aivika Experiment Chart[9]. That model depends on external random parameters and the corresponded Aivika experiment allows us to define based on the Monte-Carlo simulation how sensitive is the model to these parameters.

7.1   Table Functions

A table function operates on the `Dynamics` computation that represents a changing value x. The second argument is a table of constant pairs (x, y). The resulting computation represents y which is calculated based on the specified arguments.

There are two table functions in Aivika. The first function uses a linear interpolation. The second one is a linear stepwise function. Note that the table must be sorted by value x in the both cases.

```
module Simulation.Aivika.Dynamics.SystemDynamics

lookupDynamics :: Dynamics Double
                  -> Array Int (Double, Double)
                  -> Dynamics Double

lookupStepwiseDynamics :: Dynamics Double
                          -> Array Int (Double, Double)
                          -> Dynamics Double
```

The next example illustrates how these table functions can be used in the model definition.

7.2   Example FishBank

The Fish Bank model is distributed along with other sample models as a part of the installation package of Simtegra MapSys[3]. This model is trying to establish a relation between the amount of fish in the ocean, a number of ships with help of which this fish is caught and the profit that the ship owners could realize.

Here we are not very interested in the model itself. I just want to illustrate how the differential equations can be defined in Aivika. Please pay attention to the fact that the equations are unordered, exactly as in maths or in popular visual simulation tools.

In the model I will use two new functions which are counterparts of the standard `min` and `max` functions.

```
minDynamics :: (Ord a) => Dynamics a -> Dynamics a -> Dynamics a
maxDynamics :: (Ord a) => Dynamics a -> Dynamics a -> Dynamics a
```

The model itself is stated below. The equations are easy to read. We mix the integrals and auxiliary variables in an arbitrary order.

```
{-# LANGUAGE RecursiveDo #-}

import Data.Array

import Simulation.Aivika.Dynamics
import Simulation.Aivika.Dynamics.Simulation
import Simulation.Aivika.Dynamics.SystemDynamics
```

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

model :: Simulation Double
model =
  mdo let annualProfit = profit
          area = 100
          carryingCapacity = 1000
          catchPerShip =
            lookupDynamics density $
            listArray (1, 11) [(0.0, -0.048), (1.2, 10.875), (2.4, 17.194),
                               (3.6, 20.548), (4.8, 22.086), (6.0, 23.344),
                               (7.2, 23.903), (8.4, 24.462), (9.6, 24.882),
                               (10.8, 25.301), (12.0, 25.86)]
          deathFraction =
            lookupDynamics (fish / carryingCapacity) $
            listArray (1, 11) [(0.0, 5.161), (0.1, 5.161), (0.2, 5.161),
                               (0.3, 5.161), (0.4, 5.161), (0.5, 5.161),
                               (0.6, 5.118), (0.7, 5.247), (0.8, 5.849),
                               (0.9, 6.151), (10.0, 6.194)]
          density = fish / area
      fish <- integ (fishHatchRate - fishDeathRate - totalCatchPerYear) 1000
      let fishDeathRate = maxDynamics 0 (fish * deathFraction)
          fishHatchRate = maxDynamics 0 (fish * hatchFraction)
          fishPrice = 20
          fractionInvested = 0.2
          hatchFraction = 6
          operatingCost = ships * 250
          profit = revenue - operatingCost
          revenue = totalCatchPerYear * fishPrice
      ships <- integ shipBuildingRate 10
      let shipBuildingRate = maxDynamics 0 (profit * fractionInvested / shipCost)
          shipCost = 300
      totalProfit <- integ annualProfit 0
      let totalCatchPerYear = maxDynamics 0 (ships * catchPerShip)
      -- results --
      runDynamicsInStopTime annualProfit

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

The model can be trivially converted to the Aivika experiment[8, 9], for example, with help of which we can plot the time series for any variables and their combinations. But if you are interested in something more heavy-weight then you might want to look at the mentioned financial model, for which the deviation chart was shown on figure 2.3 in the beginning of this document.

The next chapter summarizes the methods we have considered till now.

Chapter 8

Advanced Features

We saw that the same simulation monads can describe models under different paradigms including System Dynamics, Discrete Event Simulation and Agent-based modeling. Their differences are erased. Everything is reduced ultimately to some function varying in time whatever complex the model would be. Then this function is reduced again to a new function of simulation specs.

It is amazing how well this idea suits the functional programming. The both functions are actually monads, which allows us to build complex models including the combined discrete-continuous models, or hybrid models.

Initially, Aivika was a very simple proof-of-concept library but now it provides rather sophisticated features that would be difficult to implement and use when programming in C++ or Java, for example.

Haskell is very high level programming language of general purpose and Aivika allows us to make it closer to the modeling language. It seems that there is an opposite tendency when the developers are trying to make their own modeling languages closer to the general purpose programming languages.

It is difficult to try to describe everything in the introductory document. Therefore I will only give a brief description of some topics that I have found important for the real modeling with help of Aivika.

8.1   Using Variables

The `Var` variable or its unboxed version `UVar` is like the `Ref` reference but it remembers all its values at every time point when the variable had changed. The variable has a similar interface but the variable is much slower than the reference. It is safer but slower. You should make a choice between using the variable or reference depending on your task.

At least, you cannot use the references in the differential equations but you can use the variables. It is related to the fact that these equations are not managed by the event queue. Time that flows in the equations is mostly unpredictable. Therefore every integral keeps the history of its values in the

integration time points, for example.

8.2   Using Arrays

There is no need in explicit support of arrays and lists due to the following fact: the `Dynamics` and `Simulation` types are monads and they are also instances of type class `MonadFix`. Therefore they allow using the recursive *do*-notation, which is necessary if you going to define the arrays of integrals. The arrays just work with Aivika.

You should only be careful with monadic computations and understand how they work. A common mistake is to create a meaningless computation, where a pure value is required.

At the same time, the arrays and lists are explicitly supported in the Aivika experiments[8, 9] as an available source of simulation data (namely, `Series`), although the necessary glue code is very small.

Moreover, you can define any `String` subscript for your arrays and lists when displaying the results on the chart, or when saving them in the file, if the default numeric-based subscript doesn't suite your needs.

8.3   Using Signals

A signal is a generalization of the .NET concepts of the event and `IObservable` but extended to work with dynamic systems safely, especially with those ones that are synchronized with the event queue.

The characteristic function is the following one which suspends the current discontinuous process until a signal will be received:

```
awaitSignal :: Signal a -> Process a
```

The references and variables trigger signals each time they change.

8.4   Queues LIFO and FIFO

The LIFO and FIFO queues are integrated with all the simulation monads and signals. They are implemented based on the `Resource` type.

8.5   External Parameters

The external parameters are naturally defined as the `Simulation` computations. Please refer to the documentation. Also please look at the financial model included as an example in the distributive of Aivika Experiment Chart[9].

8.6   Monte-Carlo Simulation

The Monte-Carlo simulation is inherently supported in Aivika. It is just enough to return the simulation data isolated within the `Simulation` computation. Then multiple simulation runs can be safely launched in parallel.

8.7   Simulation Experiments

The simulation experiments are implemented in packages Aivika Experiment[8] and Aivika Experiment Chart[8] which are a natural add-on to the Aivika library. They allow receiving the results of simulation as charts, tables and so on. The output of the experiment is a complete HTML page with the results. Almost every aspect can be customized. There are the deviation chart and histograms.

   The Monte-Carlo simulation is supported in these experiments.

8.8   Gathering Statistics

In the model you can gather statistics, by which then the deviation chart or histogram can be plotted. The `SamplingStats` statistics and a list of numbers can be used as a source of data for the deviation chart in a addition to other sources. Similarly, the list of numbers can be used as a source of data for the histogram in addition to other sources. The `SamplingStats` value is more efficient computationally than the list.

Bibliography

[1] Norm Matloff. *Introduction to Discrete-Event Simulation and the SimPy Language*, 2008,
http://heather.cs.ucdavis.edu/ matloff/156/PLN/DESimIntro.pdf

[2] AnyLogic Tutorial. http://www.xjtek.com/anylogic/help

[3] Simtegra MapSys. http://www.simtegra.com

[4] Vensim. http://www.vensim.com

[5] iThink. http://www.iseesystems.com

[6] The F# version of Aivika. http://sourceforge.net/projects/aivika/

[7] The Haskell version of Aivika. http://hackage.haskell.org/package/aivika

[8] The Haskell version of Aivika, facilities for providing the simulation experiments. http://hackage.haskell.org/package/aivika-experiment

[9] The Haskell version of Aivika, charts and histograms for the simulation experiments. http://hackage.haskell.org/package/aivika-experiment-chart

[10] The Scala version of Aivika. http://github.com/dsorokin/scala-aivika