An Introduction to Aivika Simulation Library

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

July 14, 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] 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 that can be prepared by Aivika as tables and charts, in a form suitable for the further more deep analysis with help of other tools such as R.

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. I checked its work 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, the creation of the integral, requesting of the value of the dynamic process at some specified time point and so on. It is important here that we can bind different computations 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.

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* in 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:

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

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]
```

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.

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 applied: 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

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.

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 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.

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
```

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.

All the rest must be updated...

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` value can play a role of the tiny time increment. 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.

We can say that Aivika supports the activity-oriented paradigm as well. But we should be cautious as this way of simulation is most risky. Below I will show how one of the considered earlier models can be coded under this paradigm and then I will show what is dangerous in that code. But before it we need some theory.

5.1 Ordered Computations and Memoization

The `Dynamics` computation by itself doesn't give any guarantee of the order of calculations. This computation corresponds to a dynamic process and we can request for its value at any time point. Therefore the computation usually contains a rule by which such a value can be calculated. It usually doesn't store the values themselves. This is a key point.

Each time we call `runDynamics1` function, the Aivika engine at first creates a model in the initial integration time point and then calls this model in the last integration time point to return the result. The `runDynamics` function requests already the model in *every* integration time point in the sequential order from the first to last with the specified integration time step.

But the `Dynamics` computation is not actually limited to a finite set of the integration time points. The computation works with the infinite set. We can request for the value at any time point from the set of real numbers. Then how does Aivika compute integrals?

The integral values are calculated in the integration time points and then interpolated for all other points. There are three predefined interpolation transformations. The integrals use the `interpolate` function.

```
module Simulation.Aivika.Dynamics.Base

initD :: Dynamics a -> Dynamics a
discrete :: Dynamics a -> Dynamics a
interpolate :: Dynamics a -> Dynamics a
```

The `initD` function returns always a value for the initial time point. It is useful if we want to know the initial value of some computation. The `discrete` function works like a linear stepwise function reducing all the time space to the integration time points only. If the requested point is different then the function returns the computation's value for the greatest integration time point not greater than the requested one. Finally, the `interpolate` function is similar to `discrete` but knows more about the integration time points.

With help of these three functions we can reduce the infinite time space to a finite space of the integration time points. The next question is how to calculate values in these points? The obvious solution is to perform the calculations sequentially starting from the initial time point to the last one with the specified integration time step. If you remember, this step is defined by the `spcDT` parameter. This is what functions `memo0` and `umemo0` do. Also they save the calculated values in the internal cache.

```
module Simulation.Aivika.Dynamics.Base

memo0 :: Dynamics e -> Dynamics (Dynamics e)

umemo0 :: MArray IOUArray e IO =>
         Dynamics e -> Dynamics (Dynamics e)
```

The latter is just an unbound version of the former. If we can create an unboxed array for some type then we can apply the last function. For example, types `Double` and `Int` are such types. Also the both functions are strict. The argument specifies a computation to memoize in the integration time points.

But the integrals need more. Some integration methods such as Runge-Kutta introduce additional steps when the same time points are used interchangeably. There are memoization functions that know of these additional steps. They are called the same only without zero on the end.

```
module Simulation.Aivika.Dynamics.Base

memo :: Dynamics e -> Dynamics (Dynamics e)
```

```
umemo :: MArray IOUArray e IO =>
        Dynamics e -> Dynamics (Dynamics e)
```

Thus, we can make the computation sequential and memoized. This is important for the integrals. The sequential order is important for the activity-oriented simulation too, although the memoization itself is somewhere redundant for this kind of simulation. Only we must guarantee that nobody else will call the computation outside the selected memo function. Usually, it is easy to provide this guarantee.

Also there is a highly optimized version of the `memo0` function for type `Dynamics ()`. The latter denotes some dynamic process with side-effects.

```
iterateD :: Dynamics () -> Dynamics (Dynamics ())
```

Now it is time of some practice.

5.2 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. Since the counters can be created in the `Dynamics` computation only, we create such a machine that returns actually a nested computation.

```
import Random
import Control.Monad.Trans

import Simulation.Aivika.Dynamics
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)

model :: Dynamics (Dynamics Double)
model =
  do queue <- newQueue
     totalUpTime <- newRef queue 0.0

     let machine :: Dynamics (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

     -- create strictly sequential computations
     c1 <- iterateD m1
     c2 <- iterateD m2

     let system :: Dynamics Double
         system =
           do c1    -- involve in the simulation
              c2    -- involve in the simulation
              x <- readRef totalUpTime
              y <- stoptime
              return $ x / (2 * y)

     return system

main =
  do a <- runDynamics1 model specs
     print a
```

To create a machine, we extract the corresponded computation from the nested one.

```
-- create two machines with type Dynamics ()

m1 <- machine
m2 <- machine
```

If you read the previous section, then you know that we cannot use these computations directly. We don't know at what time point they will be called. To order them, we can apply the `iterateD` function that does all the work.

```
-- create strictly sequential computations
c1 <- iterateD m1
c2 <- iterateD m2
```

To take effect, these new computations must be involved in the main simulation. This is what the next lines of the code do.

```
system =
  do c1    -- involve in the simulation
     c2    -- involve in the simulation
```

If we used computations `m1` and `m2` instead of `c1` and `c2` here then there would be no any simulation. Aivika would request the values of `m1` and `m2` at the last integration time point and that would be an end.

Now requesting for the value in the last time point from the `runDynamics1` function leads to a full and ordered calculation in all integration time points starting from the initial one. The `iterateD` function guarantees it. Also it is important that nobody else uses computations `m1` and `m2` expect for this memo function. It would be an error even if you used the same computation twice.

```
-- ERROR
c1  <- iterateD m1
c1' <- iterateD m1
```

Also we use the standard Aivika references, but it makes no any special sense. We could use the `IORef` references with the same success. The event queue is not used here in any way. Nothing depends on the queue. I provided such references only to attract your attention to this detail.

Thus, the activity-oriented simulation requires much manual work. Also we have to deal with an uncertain order of calculations which is inherent in the `Dynamics` computation. Compare with that how easily we could define the same model under the event-oriented and process-oriented paradigms. The event queue is a great achievement in simplifying the simulation. The next chapter shows how the same queue can be applied to model 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 downmost active state. Its ancestor line will be activated. Other states will be deactivated if required. The same ancestor can stay activated during a change of the selected state. 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 agent-based modeling. As almost everything else, this support is based on the `Dynamics` monad. The activation and deactivation procedures are the `Dynamics` computations. So are the timer and timeout handlers. As before, all is ruled by the event queue.

6.1 Stateful Agents

The agents and their states are created as part of the `Dynamics` computation. The agent is bound to the specified event queue. The state is bound to its agent. Also the state can have a parent state.

```
module Simulation.Aivika.Dynamics.Agent

data Agent
data AgentState
```

```
instance Eq Agent
instance Eq AgentState

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

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

Each agent has a selected active state. It is always a downmost 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 know this downmost 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` wrapped in the `Dynamics` monad. We can initiate the agent and select another downmost active state with help of the same function. This is function `activateState`.

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

The `initState` is very similar to the first function but it works only during the direct activation when namely this state is selected. It means that the `initState` function can be called only from the activation computation. If the state is activated implicitly when its descendant becomes active then the `initState` function is just ignored. It allows us to manage the state initialization.

Each state has the activation and deactivation computations. They are actuated if necessary. By default they are empty. We can modify them with help of the following two functions.

```
stateActivation :: AgentState -> Dynamics () -> Dynamics ()
stateDeactivation :: AgentState -> Dynamics () -> Dynamics ()
```

They look like statements that the specified state has this activation and that deactivation computations.

What makes the agent alive is the timeout and timer handlers. They are similar to events and they are indeed implemented as the events. 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 new handlers at the time of next state activation.

```
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 still actuated then it happens only once. The timer handler tries to add itself again. It will periodically repeat while the state remains active. Therefore the time 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 stays calculated. If the time period is defined as the `Dynamics` computation then it will be recalculated each time the timer handler tries to add itself again.

Like other cases the event queue manages all the process here. On the underlying level it treats the timer and timeout handlers as event handlers. Each agent state has a version number. When we add a new handler, we save the current version with the corresponded event handler. If the state becomes deactivated then its version increases, which makes all handlers with less version number obsolete. It is efficient enough.

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

6.2 Example BassDiffusion

An agent-based version of the Bass Diffusion model[2] is described in the AnyLogic 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[1] starts similarly. We import the modules, define constants, simulation specs and two random functions.

[1]If this model seems too difficult to you then I recommend to look at its more detailed version with comprehensive comments kindly prepared by Daniel Kahlenberg: https://github.com/573/abm_approaches/tree/master/aivika-hs-demo

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

import Simulation.Aivika.Dynamics
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 -> Dynamics Person
createPerson q =
  do agent <- newAgent q
     potentialAdopter <- newState agent
     adopter <- newState agent
```

```
     return Person { personAgent = agent,
                     personPotentialAdopter = potentialAdopter,
                     personAdopter = adopter }

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

Since the agents and states are created in the `Dynamics` computation, we have to separate different steps. At first step we create the objects. At second step we define their activation and deactivation computations.

```
definePerson :: Person -> Array Int Person
                -> Ref Int -> Ref Int -> Dynamics ()
definePerson p ps potentialAdopters adopters =
  do stateActivation (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'
     stateActivation (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')
     stateDeactivation (personPotentialAdopter p) $
       modifyRef potentialAdopters $ \a -> a - 1
     stateDeactivation (personAdopter p) $
       modifyRef adopters $ \a -> a - 1

definePersons :: Array Int Person
                 -> Ref Int -> Ref Int -> Dynamics ()
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 when the adopter tries to make a random agent an adopter too. Note that the time period for the timer is specified as the `Dynamics` computation. It will be recalculated during 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
```

An agent activation is straightforward enough. Each agent starts with the potential adopter state.

```
model :: Dynamics (Dynamics [Int])
model =
  do q <- newQueue
     potentialAdopters <- newRef q 0
     adopters <- newRef q 0
     ps <- createPersons q
     definePersons ps potentialAdopters adopters
     activatePersons ps
     return $ do i1 <- readRef potentialAdopters
                 i2 <- readRef adopters
                 return [i1, i2]

main =
  runDynamics model specs
```

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]]
```

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

Chapter 7

System Dynamics

A model of System Dynamics is a dynamic system with loopbacks. Usually, it is a system of differential equations (ODEs). It can have *stocks* such as *reservoirs*, *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 integration method 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 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 are easily defined in Aivika. Before we proceed to an example, I will introduce the table functions that are very useful in such equations.

7.1 Table Functions

A table function operates on the `Dynamics` computation that represents value x. The second argument is a table of 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
```

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

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

The next example illustrates how these table functions can simplify 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.

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

```
maxD :: (Ord a) => Dynamics a -> Dynamics a -> Dynamics a
minD :: (Ord a) => Dynamics a -> Dynamics a -> Dynamics a
```

The model itself is stated below. The equations are easy to read. At first we initialize the integrals, then define the auxiliaries and finally set the derivatives creating loopbacks.

```
import Data.Array

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

specs = Specs { spcStartTime = 0,
                spcStopTime = 13,
                spcDT = 0.01,
                -- spcDT = 0.000005,
                spcMethod = RungeKutta4 }

model :: Dynamics (Dynamics Double)
model =
  do fishInteg <- newInteg 1000
     shipsInteg <- newInteg 10
     totalProfitInteg <- newInteg 0
     -- integral values --
     let fish = integValue fishInteg
         ships = integValue shipsInteg
```

```
        totalProfit = integValue totalProfitInteg
    -- auxiliary values --
    let annualProfit = profit
        area = 100
        carryingCapacity = 1000
        catchPerShip =
          lookupD 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 =
          lookupD (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
        fishDeathRate = maxD 0 (fish * deathFraction)
        fishHatchRate = maxD 0 (fish * hatchFraction)
        fishPrice = 20
        fractionInvested = 0.2
        hatchFraction = 6
        operatingCost = ships * 250
        profit = revenue - operatingCost
        revenue = totalCatchPerYear * fishPrice
        shipBuildingRate =
          maxD 0 (profit * fractionInvested / shipCost)
        shipCost = 300
        totalCatchPerYear = maxD 0 (ships * catchPerShip)
    -- derivatives --
    integDiff fishInteg
      (fishHatchRate - fishDeathRate - totalCatchPerYear)
    integDiff shipsInteg shipBuildingRate
    integDiff totalProfitInteg annualProfit
    -- results --
    return annualProfit

main = do a <- runDynamics1 model specs
          print a
```

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

Chapter 8

Advanced Features

We saw that the same `Dynamics` computation can describe models under very different simulation 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. Such a function represents some underlying dynamic process. Therefore I often call the `Dynamics` computation a dynamic process. I use these terms as interchangeable.

It is amazing how well this idea suits the functional programming. This function is actually a monad. In other words, the dynamic process is a monad, which makes the former just a fantastic build unit to create simulation models. What is also important, we can mix different models together even if they were created under different paradigms. For example, we can mix agents, events, discontinuous processes and differential equations in the same hybrid model. And all this huge thing will work as one complex dynamic process, i.e. some value in the `Dynamics` monad.

Different parts of the hybrid model must communicate data to each other. If the simulation time is entirely managed by the event queue, which is the case for DES and the agent-based modeling, then it is enough to use references only. But the references has no memory, they are not bound with the time points at which they change.

Therefore we need another entity, *variables* with memory, that remember their history of changes. Then we can use the variables in such models where the time management is more difficult. For example, it takes place when we have to pass data from DES to the differential equations.

8.1 Variables with Memory

The variable is like a reference but it remembers all its values at every time point when the variable 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.

If the simulation is fully managed by a single event queue then there is a guarantee that the time flows synchronously in every part of the model. We can use references. But if we apply the activity-oriented paradigm or define the differential of difference equations then the time flows already unpredictably in general. Such equations cannot already depend on the references as the equations define their own time. We have to use the variables. However, we can still use the references in those parts, where the time is ruled by the event queue.

To create a new variable, we have to pass the event queue and initial value:

```
module Simulation.Aivika.Dynamics.Var

data Var a

newVar :: EventQueue -> a -> Dynamics (Var a)
```

This variable will depend on the event queue and the state will be synchronized with the time of this queue.

To read, modify and write a new value, we can use the following functions. They are very similar to the corresponded functions that are used with the references.

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

At any simulation time point, we can get an array that reprensents a history of changes for the specified variable. Sometimes it can be useful. The underlying array that is used by the variable itself remains unaffected.

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

The variable uses a general mutable array to store the values. For numeric types it can be inefficient. Therefore there is a subtype of the variables that uses an unboxed array. It is called `UVar`.

These new variables have a similar interface.

```
module Simulation.Aivika.Dynamics.UVar

data UVar a

newUVar :: (MArray IOUArray a IO) =>
           EventQueue -> a -> Dynamics (UVar a)

readUVar :: (MArray IOUArray a IO) =>
            UVar a -> Dynamics a
```

```
writeUVar :: (MArray IOUArray a IO) =>
             UVar a -> a -> Dynamics ()

modifyUVar :: (MArray IOUArray a IO) =>
              UVar a -> (a -> a) -> Dynamics ()

freezeUVar :: (MArray IOUArray a IO) =>
              UVar a -> Dynamics (Array Int Double, Array Int a)
```

You should prefer the unboxed version of variables whenever possible. They consume less memory and they are more efficient in general.

8.2 Using Arrays

The arrays are an inherent part of Haskell. Therefore they are naturally available in the simulation. There is no need in explicit support. Moreover, the new Vector type should be very useful as it seems that this type supports monadic computations.

8.3 Using Signals

A signal is a generalization of the .NET concept of the event but extended to work with dynamic systems.

```
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 the resources and signals.

8.5 External Parameters

The external parameters are naturally defined as the `Simulation` computations. Please refer to the documentation.

8.6 Monte-Carlo Simulation

The Monte-Carlo simulation is supported by design of Aivika. Please look at the next section, where the simulation experiments are described.

The multiple simulation runs can be launched in parallel.

8.7 Simulation Experiments

The simulation experiments (packages aivika-experiment and avika-experiment-chart) are a natural add-on to the Aivika library. They allow us to receive the results of simulations as charts, tables and so on. The output of the experiment is a complete HTML page with the results. Almost every aspect is customized.

The Monte-Carlo simulation is supported in the simulation experiments. So, there are the deviation chart and histograms.

8.8 Charts and Histograms

Some examples.

8.9 Gathering Statistics

Please refers to the documentation.

The sampling statistics can be used as a source of data for the deviation chart. The list of values can be used as a source of data for the histograms in addition to other sources.

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/nav/1_2

[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