Aivika:
A Simulation Library

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

March 28, 2011

Contents

1 Introduction **5**

2 Dynamic Systems **7**

3 Discrete Event Simulation **13**
3.1 Event Queue 13
3.2 References 14
3.3 Example MachRep1 15

4 Process-oriented Simulation **19**
4.1 Discontinuous Processes 19
4.2 Revised Example MachRep1 21
4.3 Resources 23
4.4 Example MachRep2 24
4.5 Example MachRep3 27

5 Activity-oriented Simulation **31**
5.1 Ordered Computations and Memoization 31
5.2 Example MachRep1 Again 33

6 Agent-based Modeling **37**
6.1 Stateful Agents 37
6.2 Example BassDiffusion 39

7 System Dynamics **45**
7.1 Table Functions 45
7.2 Example FishBank 46

8 Hybrid Simulation **49**

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 using the Runge-Kutta method and Euler's method. Before that I had developed a visual simulation tool Simtegra MapSys[3] together with Dr. Zahed Sheikholeslami for the field of System Dynamics.

It is 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. It is turned out that my invention can be applied to simulating the wide range of dynamic systems that evolve and change in time. So I created an F# library which I called Aivika[6]. Aivika is also a female Mari name pronounced with accent on the last syllable.

Below is described a Haskell port of my simulation library Aivika. It follows the same idea but has a slightly different API and implementation.

In chapter 2 a new monad `Dynamics` is introduced. This is a key point of my approach. A computation in this monad can be identified with some dynamic process. It is important that we can bind different computations to create new ones. It is shown how the differential equations can be written in this monad and then simulated.

Chapter 3 describes the Event Queue. The queue behaves like a coordination center processing events. It is important that the events are the `Dynamics` computations, 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. A new monad `DynamicsProc` is introduced. Only now the `DynamicsProc` computation can be identified with the discontinuous process. Such a process can be suspended at any time and then resumed later. Also the discontinuous process can behave like a dynamic process, for any `Dynamics` computation can be lifted to this new monad. So we can mix computations in the both monads. It allows us to simulate the models under the process-oriented paradigm of DES.

Chapter 5 returns us to the `Dynamics` monad. It shows how we can simulate the models under the activity-oriented paradigm of DES. Also this chapter

uncovers some internals of the `Dynamics` computation.

Chapter 6 shows how the approach can be applied to 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. It is shown how the approach can be applied to System Dynamics.

The last chapter 8 summarizes my approach and is devoted to hybrid models stating that the method is actually open to new simulation techniques.

Chapter 2

Dynamic Systems

A *dynamic system* evolves and changes in time. Examples are systems of ordinary differential equations with help of which we can describe the models of System Dynamics. Each time we define a Discrete Event Simulation task, we also define a time varying dynamic system. Finally, in the Agent-based Modeling we define agents and 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 also a notion of *dynamic process*. This is a generalization of the time-varying function. The process can take any values of some arbitrary set. We don't obligate the process to take only numeric values in time points. Complex data structures can also be values of the dynamic process. Therefore it is natural that the dynamic process can be represented as a monad. In the Aivika simulation library this is the `Dynamics` monad.

```
data Dynamics a

instance Functor Dynamics
instance Monad 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 represents some dynamic process that varies in time. This process can take numerical values. Moreover, we can construct mathematical expressions from such processes, for this monad can be an instance of 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 the time parameters to every part of the imperative computation.

There are four primitives that allow us to know the time parameters:

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

The `starttime` value represents the initial time of the simulation. The `stoptime` value gives us the information about the final time of the simulation. The `dt` value 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 value returns the current simulation time.

Having only these definitions, we can define interesting dynamics 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 * sin (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 other data as well, for example, integrals. In Aivika the *integral* is some structure that has an initial value, current value and the time derivative.

```
data Integ

newInteg :: Dynamics Double -> Dynamics Integ
integValue :: Integ -> Dynamics Double
integDiff :: Integ -> Dynamics Double -> Dynamics ()
```

Here the `newInteg` function creates a new integral with the specified initial value. The `integValue` function returns the current integral value. The `integDiff` function allows us to set the derivative of the integral. The returned type of the last function means that we perform some side effect during the

computation. We actually create a loopback updating some reference in the `IO` monad.

Now we can define the system of ordinary differential equations (ODE). The idea is as follows. At first we create integrals, then we define all equations but the derivatives. To refer to the integrals, we use the `integValue` function. At last we complete the system by setting the derivatives. This scheme allows us to define complex systems with loopbacks.

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:

```
model :: Dynamics (Dynamics [Double])
model =
  do integA <- newInteg 100
     integB <- newInteg 0
     integC <- newInteg 0
     let a = integValue integA
         b = integValue integB
         c = integValue integC
         ka = 1
         kb = 1
     integDiff integA (- ka * a)
     integDiff integB (ka * a - kb * b)
     integDiff integC (kb * b)
     return $ sequence [time, a, b, c]
```

Please note an interesting thing. We return a nested dynamic process. The model is some process that creates a simulation. To integrate the differential equations, we have to allocate some memory and create loopbacks. This is what the outer process does. The inner process is the simulation itself. This is so common in Aivika that all run functions for the `Dynamics` monad take a nested dynamic process as their argument.

```
runDynamics1 :: Dynamics (Dynamics a) -> Specs -> IO a
runDynamics :: Dynamics (Dynamics a) -> Specs -> IO [a]
runDynamicsIO :: Dynamics (Dynamics a) -> Specs -> IO [IO a]
```

The `runDynamics1` function creates a simulation and then runs it in the last integration time point using the specified simulation specs. Similarly, the

`runDynamics` function creates a simulation and then runs it in all integration time points using the specified simulation specs. Like the previous one, the `runDynamicsIO` function does the same and runs the simulation in all integration time points but returns raw data in the IO monad.

Using any of the run functions, we can simulate the model. All them require the *simulation specs* that have the following definition with the obvious meaning:

```
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 is namely that thing which provides the `starttime`, `stoptime`, `dt` and `time` built-in values considered above with the input data. The integration method has effect only on the integrals. All other dynamic processes just ignore it.

There is a subtle thing related to the `Dynamics` monad and function `join`. A computation in the monad returns multiple values calculated in time points. It was tempting to write ordinary run functions without the nested type like that

```
-- N.B. this function doesn't exist.
runDynamics' :: Dynamics a -> Specs -> IO [a]

-- ERROR: this is wrong!
main = do xs <- runDynamics' (join model) specs
          print xs
```

But that might lead to incorrect results. We could not use the same cache for the integrals. This cache would be recreated for each time point during the simulation, which would be impractical and even wrong in some cases!

Therefore it is important that the run functions actually take a nested type that corresponds to a dynamic process inside another process. The outer process creates once a cache and other intermediate data such as references. It happens in the initial integration time point before the actual simulation is started and when the information about the simulation specs becomes available. Then the inner process can use already these shared data many times in different time points to simulate the model.

> In the F# version of Aivika the run functions use an unnested argument like the `runDynamics'` function from the example above. It is possible due to impurity and eager evaluation. Moreover, we can create new integrals calling an ordinary object constructor. Some

side effects are hidden. It is not the case for the Haskell version of Aivika, where all side effects are explicit.

Now we can simulate our ODE system.

```
main =
  do xs <- runDynamics model specs
     print xs
```

Let the initial time be 0, final time be 10, integration time step equal 1 and we apply the 4th order Runge-Kutta method.

```
specs = Specs { spcStartTime = 0,
                spcStopTime = 10,
                spcDT = 1,
                spcMethod = RungeKutta4 }
```

The simulation will return the following results:

```
[[0.0,100.0,0.0,0.0],
 [1.0,37.5,33.33333333333333,29.166666666666664],
 [2.0,14.0625,25.0,60.9375],
 [3.0,5.2734375,14.0625,80.6640625],
 [4.0,1.9775390625,7.03125,90.9912109375],
 [5.0,0.7415771484375,3.2958984375,95.9625244140625],
 [6.0,0.2780914306640625,1.483154296875,98.23875427246094],
 ...]
```

We saw that the dynamic processes and systems can be modeled with help of the `Dynamics` monad. In continuation of this subject the next chapter shows how this monad 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 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 called *event-driven*.

3.1 Event Queue

If the `Dynamics` monad is a circulatory system of Aivika then the event queue is its heart. The queue is a motor that makes a model alive. Very much in Aivika depends on the event queue. Its interface is very simple.

```
data DynamicsQueue

newQueue :: Dynamics DynamicsQueue

enqueueD :: DynamicsQueue -> Dynamics Double
            -> Dynamics () -> Dynamics ()
enqueueD q t m
```

The `newQueue` function creates a new *event queue*. It performs some side effect. Therefore the result is wrapped in the monad. The `enqueueD` function is rather interesting. It adds event `m` to queue `q`. The event must be raised at time `t`. The result is a computation. The most exciting thing is that the event is a computation too in the `Dynamics` monad. If we want to pass some message with the event then we should use a closure.

So, the *event* is a dynamic process that has a single purpose to perform once some side effect at the specified time. The time argument also takes a form of the process but it was made mostly for easiness. We can create expressions using the time built-in values to define the moment at which the event will be raised.

To functionate properly, the event queue must be involved in the main simulation. The next function returns a computation that represents a moving force of the queue. The resulting computation should be added to the model, although many objects you will see later do it implicitly.

```
runQueue :: DynamicsQueue -> Dynamics ()
```

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

In the F# version of Aivika the event queue has type `Env`.

Before we proceed to the simulation approach, I will introduce a reference type that can be applied to store and pass some data between different parts of the model.

3.2 References

Values of the `DynamicsReference` type are very similar to values of the standard `IORef` type except for one thing. Each of them is bound to some event queue. Before the reference value is requested the corresponded queue is checked whether there are pending events that should be raised. It makes the model coordinated. If you bind the reference to the event queue then you receive a guarantee that all that depends on this queue will be actual at time of requesting for the reference value.

```
data DynamicsRef a

newRef :: DynamicsQueue -> a -> Dynamics (DynamicsRef a)

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

Because of laziness, in general you should not use functions `writeRef` and `modifyRef` in the simulation, for they create thunks that can lead to a space leak. There are their strict counterparts that you should use, especially if the reference stores a numeric value.

```
writeRef' :: DynamicsRef a -> a -> Dynamics ()
modifyRef' :: DynamicsRef a -> (a -> a) -> Dynamics ()
```

Thus, with help of 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 a single event queue in the whole model, because of the performance consideration too.

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.

As before, we create a model that computes the simulation. Also we need an auxiliary function to generate exponentially distributed random values. To run the simulation, we define the specs which some values are rather conditional. We just have to define them anywhere.

```
import Random
import Control.Monad.Trans

import Simulation.Aivika.Dynamics

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 :: Dynamics (Dynamics 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 = return $ finishUpTime + repairTime
             enqueueD queue t machineRepaired

        machineRepaired :: Dynamics ()
        machineRepaired =

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

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

    -- start the first machine
    enqueueD queue starttime machineRepaired

    -- start the second machine
    enqueueD queue starttime machineRepaired

    let system :: Dynamics Double
        system =
          do x <- readRef totalUpTime
             y <- stoptime
             return $ x / (2 * y)

    return system

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

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 `runQueue` function, which is called implicitly each time we call the `readRef` function in the main simulation loop.

Here the events are created by two local functions `machineBroken` and `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
enqueueD queue starttime machineRepaired

-- start the second machine
enqueueD queue starttime machineRepaired
```

Here we initialize the event queue passing 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.

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 the start up time with the closure.

```
-- enqueue a new event
let t = return $ startUpTime + upTime
enqueueD 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 = return $ finishUpTime + repairTime
enqueueD queue t machineRepaired
```

During the repair time we update our counter using the strict function `modifyRef'`. If we used a non-strict version that we would achieve a space leak. This is a rather subtle thing.

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

Finally, we call the simulation at the last integration time point using the `runDynamics1` function. It calls at that time point the `readRef` function that in its turn unwinds all the events starting from the initial integration time point, for the reference is bound to 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 suspend and resume such processes. Also we can request for and release of the resources suspending and resuming the processes in case of need.

Before we proceed to examples, I have to introduce this kind of dynamic processes.

4.1 Discontinuous Processes

Aivika provides a special kind of dynamic processes which I will call *discontinuous processes* to distinguish them from the dynamic processes that have type `Dynamics`. They are important for the process-oriented simulation. These processes of the new kind can be suspended at any time and then resumed later. It allows us to model better the corresponded activities.

So, a discontinuous process is a value of type `DynamicsProc`. In most cases it can behave like the dynamic process. Indeed, any computation of type `Dynamics` can be lifted to the `DynamicsProc` type.

```
data DynamicsProc a

class DynamicsTrans m where
  liftD :: Dynamics a -> m a

instance DynamicsTrans DynamicsProc
```

It allows us to include other dynamic processes in the computation of the discontinuous process. For example, expression `liftD time` returns a current simulation time wrapped in this new type.

Moreover, the `DynamicsProc` type is a monad, which is very important to define the models.

```
instance Functor DynamicsProc
instance Monad DynamicsProc
instance MonadIO DynamicsProc
```

Tha main characteristic of the discontinuous processes is their ability to suspend. Each of the next two functions suspend the current computation in the `DynamicsProc` monad for the specified time. The time can be either a number or a value wrapped in the `Dynamics` monad.

```
holdProcD :: Dynamics Double -> DynamicsProc ()
holdProc :: Double -> DynamicsProc ()
```

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

```
passivateProc :: DynamicsProc ()
```

The difference between a hold and passivation is that the hold process stops and adds an awakening event to the underlying 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 PID*.

The process PID is actually a handle. Each process is bound to 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.

```
data ProcPID
instance Eq ProcPID

procPID :: DynamicsProc DynamicsPID
```

To reactivate another process, we must know its PID. Also we can test whether a process with the specified PID is passivated, or not. The next two functions don't affect the current computation.

```
reactivateProc :: DynamicsPID -> DynamicsProc ()
procPassive :: DynamicsPID -> DynamicsProc Bool
```

A time of the process PID creation and a time of the discontinuous process start are separated. It allows us to create PIDs, define some logic of the processes that use these PIDs and then already launch the processes.

```
newPID :: DynamicsQueue -> Dynamics DynamicsPID
runProc :: DynamicsProc () -> DynamicsPID -> Dynamics Double
           -> Dynamics ()
```

The `newPID` function requires an event queue. This queue acts behind the scene each time we hold a process for the specified time or reactivate the previously passivated process.

The `runProc` function starts the discontinuous process at the specified time. Also we must assign a PID to the new process. Note that the PIDs must be unique. Already used PID cannot be used again.

> In the F# version of Aivika I have implemented a slightly different scheme. There is an object instance that stores the process continuation. This object is identified with the process itself. Instead of the `DynamicsProc` monad the F# version uses a more simple `DynamicsCont` monad. Actually, the Haskell version also uses this monad underneath the `DynamicsProc` monad.

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 from section 3.3 can be rewritten using the discontinuous processes. Below I give the problem statement[1] 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.

The main idea is to represent each of the machines as a separate discontinuous process, i.e. a computation in the `DynamicsProc` monad.

```
import Random
import Control.Monad.Trans

import Simulation.Aivika.Dynamics

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

     pid1 <- newPID queue
     pid2 <- newPID queue

     let machine :: DynamicsProc ()
         machine =
           do startUpTime <- liftD time
              upTime <- liftIO $ exprnd upRate
              holdProc upTime
              finishUpTime <- liftD time
              liftD $ modifyRef' totalUpTime
                (+ (finishUpTime - startUpTime))
              repairTime <- liftIO $ exprnd repairRate
              holdProc repairTime
              machine

     runProc machine pid1 starttime
     runProc machine pid2 starttime

     let system :: Dynamics Double
         system =
           do x <- readRef totalUpTime
              y <- stoptime
              return $ x / (2 * y)

     return system

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

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, i.e. when `time` becomes greater than

`stoptime`. 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 computation which has another type `DynamicsProc`. Therefore we apply the `liftD` function. In such a way we get to know of the current simulation time inside of the `DynamicsProc` computation.

```
do startUpTime <- liftD 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.

```
holdProc upTime
```

In the specified time the process will be resumed and its control flow will continue. Then we update the counter, calculate the repair time and suspend the process again. After the process is resumed at the second time we repeat all calling the process computation recursively.

To initiate two separate processes at the start time of simulation, we use the `runProc` function.

```
runProc machine pid1 starttime
runProc machine pid2 starttime
```

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

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

4.3 Resources

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

```
data DynamicsResource
instance Eq DynamicsResource

newResource :: DynamicsQueue -> Int -> Dynamics DynamicsResource
```

The event queue is necessary 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 in the current computation of the discontinuous process.

```
requestResource :: DynamicsResource -> DynamicsProc ()
```

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 :: DynamicsResource -> DynamicsProc ()
```

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 know the available count of the limited resource, we can call function `resourceCount`. The next 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 :: DynamicsResource -> DynamicsProc Int
resourceInitCount :: Int
resourceQueue :: DynamicsQueue
```

This small set of the new functions allows us to build models with more complex behavior.

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

```
import Simulation.Aivika.Dynamics

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 :: Dynamics (Dynamics (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 <- newPID queue
     pid2 <- newPID queue

     let machine :: DynamicsProc ()
         machine =
           do startUpTime <- liftD time
              upTime <- liftIO $ exprnd upRate
              holdProc upTime
              finishUpTime <- liftD time
              liftD $ modifyRef' totalUpTime
                (+ (finishUpTime - startUpTime))

              -- check the resource availability
              liftD $ modifyRef' nRep (+ 1)
              n <- resourceCount repairPerson
```

```
              when (n == 1) $
                liftD $ modifyRef' nImmedRep (+ 1)

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

              machine

     runProc machine pid1 starttime
     runProc machine pid2 starttime

     let system :: Dynamics (Double, Double)
         system =
           do x <- readRef totalUpTime
              y <- stoptime
              n <- readRef nRep
              nImmed <- readRef nImmedRep
              return (x / (2 * y),
                      fromIntegral nImmed / fromIntegral n)

     return system

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

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.

```
liftD $ modifyRef' nRep (+ 1)
n <- resourceCount repairPerson
when (n == 1) $
  liftD $ 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 PID of another machine. We pass it as a parameter.

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

import Simulation.Aivika.Dynamics

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 :: Dynamics (Dynamics 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 <- newPID queue
     pid2 <- newPID queue

     let machine :: DynamicsPID -> DynamicsProc ()
         machine pid =
           do startUpTime <- liftD time
              upTime <- liftIO $ exprnd upRate
              holdProc upTime
              finishUpTime <- liftD time
              liftD $ modifyRef' totalUpTime
                (+ (finishUpTime - startUpTime))

              liftD $ modifyRef' nUp $ \a -> a - 1
              nUp' <- liftD $ readRef nUp
              if nUp' == 1
                then passivateProc
                else do n <- resourceCount repairPerson
                        when (n == 1) $ reactivateProc pid

              requestResource repairPerson
              repairTime <- liftIO $ exprnd repairRate
              holdProc repairTime
              liftD $ modifyRef' nUp $ \a -> a + 1
              releaseResource repairPerson

              machine pid

     runProc (machine pid2) pid1 starttime
```

```
    runProc (machine pid1) pid2 starttime

    let system :: Dynamics Double
        system =
          do x <- readRef totalUpTime
             y <- stoptime
             return $ x / (2 * y)

    return system

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

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.

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

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

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

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

```
runProc (machine pid2) pid1 starttime
runProc (machine pid1) pid2 starttime
```

The process-oriented simulation is built on top of the event queue and the `Dynamics` computation, i.e. on top of the event-driven simulation. But we can actually create models based on the `Dynamics` computation directly even without queue, although it is more risky as we loose the coordination center in the form of the event queue. The next chapter is devoted to this subject.

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.

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

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. Also it takes into account an accuracy. Finally, the `interpolate` function is similar to `discrete` but applies a linear interpolation between the closest 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 `memo0'` do. Also they save the calculated values in the internal cache.

```
memo0 :: Memo e => (Dynamics e -> Dynamics e)
        -> Dynamics e
        -> Dynamics (Dynamics e)

umemo0 :: UMemo e => (Dynamics e -> Dynamics e)
         -> Dynamics e
         -> Dynamics (Dynamics e)

instance Memo e
```

The latter is just an unbound version of the former. Two type classes `Memo` and `UMemo` are used here. We need them to create mutable arrays in which the computation's values are stored. Any type is an instance of the `Memo` type class. The `UMemo` class is more restrictive. If we can create an unboxed array for some type then this type is an instance of the `UMemo` class. For example, `Double` and `Int` are instances of the both classes. Also the both functions are strict.

The first argument must be an interpolation function. The second argument specifies a computation to memoize in the integration time points. In all other points the values are interpolated using the specified interpolation function.

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.

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

umemo :: UMemo e => (Dynamics e -> Dynamics e)
        -> 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.

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

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 <- memo0 discrete m1
    c2 <- memo0 discrete 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 `memo0` function, although the caching itself will be redundant here.

```
-- create strictly sequential computations

c1 <- memo0 discrete m1
c2 <- memo0 discrete 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 `memo0` 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  <- memo0 discrete m1
c1' <- memo0 discrete 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.

```
data Agent
data AgentState

instance Eq Agent
```

```
instance Eq AgentState

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

agentQueue :: Agent -> DynamicsQueue
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. It 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.

```
addTimeoutD :: AgentState -> Dynamics Double -> Dynamics ()
               -> Dynamics ()
addTimerD :: AgentState -> Dynamics Double -> Dynamics ()
             -> Dynamics ()

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

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 events. Each agent state has a version number. When we add a new handler, we save the current version with the corresponded event. 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 starts similarly. We import the modules, define constants, simulation specs and two random functions.

```
import Random
import Data.Array
```

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

import Simulation.Aivika.Dynamics

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 be an adopter too.

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

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

createPersons :: DynamicsQueue -> 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
                -> DynamicsRef Int -> DynamicsRef 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!
          addTimerD (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
                 -> DynamicsRef Int
                 -> DynamicsRef 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 =
  do xs <- runDynamics model specs
     print xs
```

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.

```
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

specs = Specs { spcStartTime = 0,
                spcStopTime = 13,
                spcDT = 0.01,
                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 xs <- runDynamics model specs
          print xs    -- N.B. it is a long output!
```

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

Chapter 8

Hybrid Simulation

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.

I would like to end this document saying that this subject is not yet completed. I expect that new simulation techniques can be developed using the method I have invented and implemented in Aivika.

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/