Aivika:
A Functional Programming Approach to Simulation in F#

David Sorokin <david.sorokin@gmail.com>

September 15, 2010

Contents

Chapter 1

Introduction

Aivika is a female Mari name pronounced with accent on the last syllable. Aivika is also a comprehensive .NET simulation library which I invented during my study of the functional programming. The library allows you to use System Dynamics, Discrete Event Simulation (DES) and the agent-based modeling in your models. Also you can create activity-oriented, event-oriented and process-oriented discrete simulations. Moreover, you can create hybrid models based on all these paradigms. I have invented a common unified scheme of the modeling. By itself this is not a new event, for there is already an excellent software tool AnyLogic[1] that allows you to create and simulate hybrid models. But what can be new is a simplicity of the library and an easiness of its integration with high-level general-purpose programming languages with help of which you can write .NET applications.

All began in November 2009 when I implemented a simple *monad* in the Haskell programming language to integrate the system of differential equations with help of Euler's method and the Runge-Kutta method. It was a working prototype of the `Dynamics` workflow described in this book. Then in December 2009 I transfered it to F#, about which I wrote a message in my Russian blog[2] in the beginning of February 2010. Then I published this information in my English blog[3] and on the main forum[4] of the F# developers. At the same time I created project Aivika[5] on SourceForge and put the first version of the library.

It wasn't a new experience for me in System Dynamics. Earlier I developed simulation tools Simtegra MapSys[6] and MapSim[7] together with Dr. Zahed Sheikholeslami. So, I knew well how such simulation tools could be implemented. Also I studied some DES models in the University. What was new for me was the agent-based modeling.

[1] http://www.xjtek.com/
[2] http://dsorokin.blogspot.com/2010/02/blog-post.html
[3] http://dsorokin-en.blogspot.com/2010/02/dynamics-modeling-monad.html
[4] http://cs.hubfs.net/forums/thread/13043.aspx
[5] http://sourceforge.net/projects/aivika/
[6] http://www.simtegra.com/
[7] http://sourceforge.net/projects/mapsim/

In March 2010 I got to know about SimPy[8], an excellent simulation library written in Python and implementing the process-oriented paradigm of DES. It was fantastic but I almost instantly remembered that *continuations* of the functional programming would allow me to achieve the same result. So, I created another monad, aka the `DynamicsCont` workflow.

In April 2010 I made publicly available the next version of Aivika which already supported System Dynamics and three paradigms of DES. Also I transfered a few models described in a wonderful Russian book by Ilya Trub. It gave me a confidence that Aivika can be a useful practical tool. I intentionally implemented some of those models as hybrid ones and it worked!

In May 2010 I read one interesting Russian article by Andrei Borshchev, one of the creators of AnyLogic. The introductory article describes different simulation paradigms but it is focused mainly on the agent-based modeling. So, in that month I created a new version of Aivika that supported the agent-based modeling as well. Also it was a significantly optimized version, where the module of System Dynamics was in 5 times faster than it was in the previous version. The new version had number 2.0.

Now Aivika supports three very different paradigms of simulation. It turned out that all them can be united on basis of monads that came from Haskell and mathematics in the world of ordinary programming. I think that Aivika is an excellent tool for fast prototyping of complex, probably hybrid, models. Aivika supports the *discrete-continuous simulation*. Also it can work with agents. These capabilities have a drawback. The simulation is slow and it essentially relies on the effective memory management system. The simulation speed was not my goal and I think that it can be rather sufficient in many cases taking into account the recent progress in the software and hardware of modern computers.

One of the main features of the monads is that they allow combining different code together. The monad can be thought as some abstract computation and its *bind* function allows us to create a manageable chain of such computations. For Aivika it means that the library can be easily integrated with .NET applications. The integration can be in the both directions. We can call external functions within the simulation. Also we can run simulations from the external .NET code. Here I think that Aivika is a promising tool for creating visualized simulations, including web-based Silverlight simulations that could be downloaded from the Internet and then played in the browser.

I hope that you will enjoy my library Aivika and that the represented book will help you in this. Here I suppose that the reader is already familiar with basics of System Dynamics, Discrete Event Simulation and the agent-based modeling. Also I suppose that you are familiar with the concept of monads at least in scopes of the Haskell programming language. Understanding monads is crucial for understanding methods of Aivika, its particular approach of building hybrid models.

[8] http://simpy.sourceforge.net/

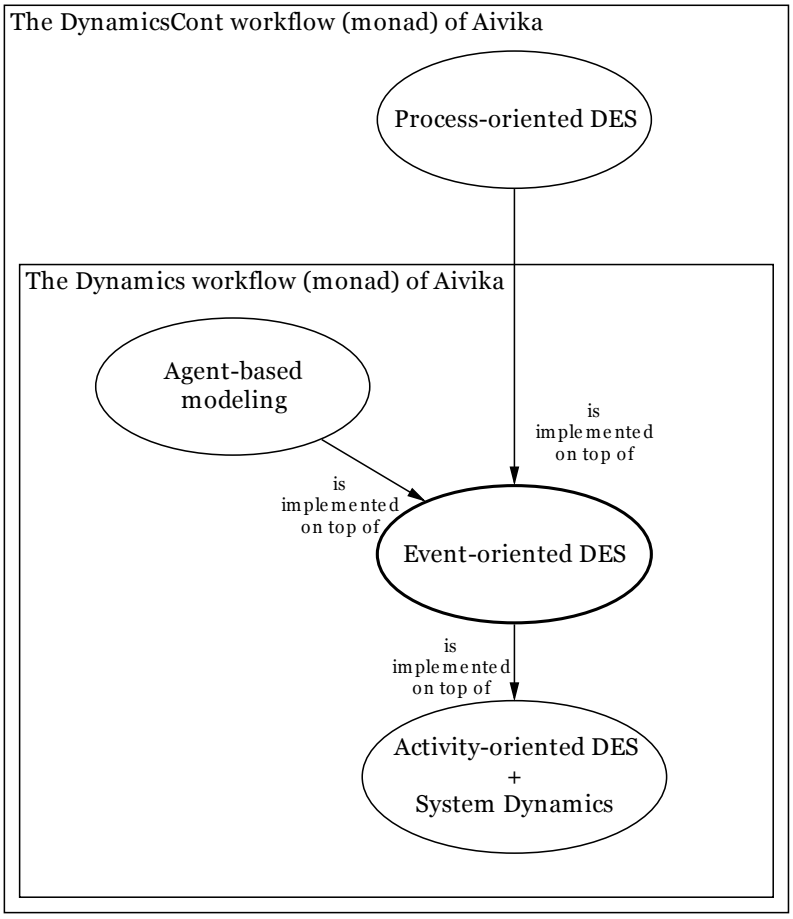

Figure 1.1: Unified Scheme of Simulation

Chapter 2

The Aivika Simulation Library

All the described below in this book is implemented in my project Aivika, which I initially created on SourceForge.net in February 2010:

http://sourceforge.net/projects/aivika/.

The project is released under the BSD license. The main target audience are the F# developers, although the module of System Dynamics can be used from C# as well. The corresponded function names are capitalized in the DLL file to look C#-friendly. In F# these names look as they are described in this book.

I tested the library on Windows under .NET and on Linux under Mono. In some tests the both frameworks show the same speed. But the process-oriented simulation (DES) tends to be significantly faster under .NET. Perhaps it is related to TCO (tail-call optimization), which is better supported by .NET CLR.

I think that Aivika should work with Silverlight as well, although I didn't test it. To generate strong pseudo-random numbers, Aivika uses one standard cryptographic module. It may cause some problems on Silverlight. In case of need this dependency can be removed as there is also a more simple random number generator.

Chapter 3

Dynamics Workflow Basics

In the heart of Aivika lies the Dynamics workflow. It identifies a computation of values of some dynamic process varying in time. All in this book is based on this workflow whether we use System Dynamics, discrete event simulation or agent-based modeling. Everywhere the Dynamics workflow performs a key role of glue that joins very different parts. As a result, it allows us to create and simulate hybrid models. At the same time the Dynamics workflow remains very easy-to-use.

3.1 Getting Started with Dynamics Workflow

The Dynamics workflow defines a computation of values of some dynamic process. The value of the corresponded computation has type `Dynamics<'a>`. One value of this type implies a whole set of other values of type `'a` that define the process. The `Dynamics` module of Aivika contains two important functions that allow us to run any dynamic process and receive its values by the specified simulation specs:

Table 3.1: Run Functions

Function and type	Description
`Dynamics.run: Dynamics<'a> -> Specs -> seq<'a>`	Runs the computation using the specified simulation specs and then returns the values in all integration nodes
`Dynamics.runLast: Dynamics<'a> -> Specs -> 'a`	Runs the computation using the specified simulation specs and then returns the value in the last integration node

The `run` function returns the values in all integration nodes. The `runLast`

function returns only the value in the last node. The both functions accept two arguments. The first argument can be any dynamic process. The second one defines the simulation specs. These specs define the start time of simulation, stop time, integration time step, the method of integration and the method of generating random values. All them are defined by a value of type `Specs`:

```
type Method = Euler | RungeKutta2 | RungeKutta4
type Randomness = SimpleRnd | StrongRnd

type Specs = {

    StartTime: float; StopTime: float; DT: float;
    Method: Method; Randomness: Randomness
}
```

The `Method` type has the obvious values which define Euler's method, the 2nd and 4th order Runge-Kutta methods respectively. The `Randomness` type has two values, one of which defines a fast but weak method of generating the pseudo-random values while the last value corresponds to a more strong but slow method.

In examples of this chapter I will use the following simulation specs:

```
>open Maritegra.Aivika;;
>let specs = {

     StartTime=0.0; StopTime=10.0; DT=1.0;
     Method=RungeKutta4; Randomness=StrongRnd
 };;

val specs : Specs = {StartTime = 0.0;
                     StopTime = 10.0;
                     DT = 1.0;
                     Method = RungeKutta4;
                     Randomness = StrongRnd;}
```

The simplest computation is that one which returns always the same value, i.e. constant. The `Dynamics` module contains the `eta` function which creates such a computation by the specified input value:

Table 3.2: Eta Function

Function and type	Description
`Dynamics.eta: 'a -> Dynamics<'a>`	Returns a computation for the specified value

This function is polymorphic. Sometimes such a function is also called generic. Here it means that the function will take a value of any type and return the corresponded computation. We can test it in the F# Interactive. The dynamic process must return the same values.

```
> let d1 = Dynamics.eta 3;;

val d1 : Dynamics<int>

> Dynamics.run d1 specs |> Seq.toList;;
val it : int list = [3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3]

> let d2 = Dynamics.eta ("Hi", [0, 1]);;

val d2 : Dynamics<string * (int * int) list>

> Dynamics.run d2 specs |> Seq.toList;;
val it : (string * (int * int) list) list =
  [("Hi", [(0, 1)]); ("Hi", [(0, 1)]); ("Hi", [(0, 1)]);
   ("Hi", [(0, 1)]); ("Hi", [(0, 1)]); ("Hi", [(0, 1)]);
   ("Hi", [(0, 1)]); ("Hi", [(0, 1)]); ("Hi", [(0, 1)]);
   ("Hi", [(0, 1)]); ("Hi", [(0, 1)])]
```

Please note that in the last case we define a dynamic process that returns a tuple of string and list. Although in the course of this book we'll use numeric simulations, it is important to remember that the `eta` function accepts an argument of any type and that the dynamic process can also return values of any type. We'll constantly use this fact to define complex dynamic systems, because the dynamic system can be considered as a more complex dynamic process that returns lists, or tuples, or even objects instead of ordinary values. We'll see it soon.

The `eta` function is so important and so widely used that Aivika defines its synonym in module `SD`, which can be opened if you write in your code `open Maritegra.Aivika.SD`. Then all functions from this module can be referenced to without direct indicating the module name. It means that you can write just `eta` in your code.

3.2 Using Arithmetic Operators

Arithmetic operators are overloaded for values of type `Dynamics<float>`. Moreover, you can mix these values with floating point numbers. The result will have type `Dynamics<float>`.

```
> let d3 = 5.0 + (eta 2.0) * (eta 3.0);;

val d3 : Dynamics<float>
```

```
> Dynamics.run d3 specs |> Seq.toList;;
val it : float list =
  [11.0; 11.0; 11.0; 11.0; 11.0;
   11.0; 11.0; 11.0; 11.0; 11.0; 11.0]
```

It allows us to write equations in a simple readable form. The same dynamic process `d3` could be also defined in the following way:

```
let d3 =
    let a = eta 2.0
    let b = eta 3.0
    let c = 5.0 + a * b
    in c
```

3.3 Using Mathematical Functions

Like the arithmetic operators the following mathematical functions are overloaded for values of type `Dynamics<float>`: `abs`, `acos`, `asin`, `atan`, `atan2`, `ceil`, `exp`, `floor`, `truncate`, `round`, `log`, `log10`, `sqrt`, `cos`, `cosh`, `sin`, `sinh`, `tan` and `tanh`. Also the binary operator `( ** )` is overloaded as well.

```
> let d4 = eta 2.0
  let d5 = (cos d4)**d4 + (sin d4)**d4;;

val d4 : Dynamics<float>
val d5 : Dynamics<float>

> Dynamics.run d5 specs |> Seq.toList;;
val it : float list =
  [1.0; 1.0; 1.0; 1.0; 1.0; 1.0; 1.0; 1.0; 1.0; 1.0; 1.0]
```

3.4 Using Computation Expression Syntax

The Dynamics workflow has builder `dynamics`. It allows us to create sophisticated computations of any complexity. Actually, the introduced before `eta` function is a synonym of the `Return` method of this builder.

```
> let map f m = dynamics {
    let! x = m
    return (f x)
 };;

val map : ('a -> 'b) -> Dynamics<'a> -> Dynamics<'b>
```

```
> let d6 = d5 |> map (fun x -> x + 1.5);;

val d6 : Dynamics<float>

> Dynamics.run d6 specs |> Seq.toList;;
val it : float list =
  [2.5; 2.5; 2.5; 2.5; 2.5; 2.5; 2.5; 2.5; 2.5; 2.5; 2.5]
```

The defined above function `map` is important. For example, it can be applied for numerical integration of mathematical functions as it will be shown further in section 4.5, page 28. Therefore function `map` is a part of the `Dynamics` module.

Table 3.3: Map Function

Function and type	Description
`Dynamics.map: f:('a -> 'b) -> m:Dynamics<'a> -> Dynamics<'b>`	Maps values of computation `m` using transformation function `f`

The computation expression is the main way of calling external .NET functions within your simulation. But you should carefully call them. If your function is pure, i.e. has no assignment nor any other side-effect, then you can call such a function at almost any place. But if your external function has a side effect, then in general you should avoid calling it from the computation expression. The Dynamics workflow usually means a delayed computation. No computation order is implied. In general case it is unknown at which moment and in which order the computation will be applied. However, there are workarounds. See section 4.13, page 45, to know how you can use your .NET functions within the simulation.

Chapter 4

System Dynamics

The models of System Dynamics can represent complex feedback systems varying in time. The Dynamics workflow allows us to define and simulate such models. The main advantage of this approach is that the model can be defined declaratively in a notation closed to mathematical. Aivika supports a wide range of ready-to-use functions which are standard and can be found in many software simulation tools such as Simtegra MapSys[1], Vensim[2] and ithink[3].

4.1 Getting Started with Differential Equations

The `SD` module contains the `integ` function that returns an integral by the specified derivative and initial value.

Table 4.1: Integral Function

Function and type	Description
`integ: d:Lazy<Dynamics<float>> -> i:Dynamics<float> -> Dynamics<float>`	Returns the integral of rate `d` and initial value `i`

The derivative must be defined as a delayed value, which allows us to declare recursively the differential equations with loopbacks as shown further.

```
> open Maritegra.Aivika;;
> open Maritegra.Aivika.SD;;
> let rec a = integ (lazy (- ka * a)) (eta 100.0)
```

[1] http://www.simtegra.com
[2] http://www.vensim.com
[3] http://www.iseesystems.com

```
  and b = integ (lazy (ka * a - kb * b)) (eta 0.0)
  and c = integ (lazy (kb * b)) (eta 0.0)
  and ka = 1.0
  and kb = 1.0;;

val a : Dynamics<float>
val b : Dynamics<float>
val c : Dynamics<float>
val ka : float = 1.0
val kb : float = 1.0
```

This system describes a simple chemical reaction. In the language of mathematics these equations look like:

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

Now we can simulate the model and, for example, return values of variable `a` in the specified integration nodes. For simplicity I will use a large enough integration time step for this task.

```
>let specs = {

     StartTime=0.0; StopTime=10.0; DT=1.0;
     Method=RungeKutta4; Randomness=StrongRnd
 };;

val specs : Specs = {StartTime = 0.0;
                     StopTime = 10.0;
                     DT = 1.0;
                     Method = RungeKutta4;
                     Randomness = StrongRnd;}

> Dynamics.run a specs |> Seq.toList;;
val it : float list =
  [100.0; 37.5; 14.0625; 5.2734375; 1.977539063; 0.7415771484;
    0.2780914307; 0.1042842865; 0.03910660744; 0.01466497779;
    0.005499366671]
```

But we often need more information than just one variable. Therefore the `Dynamics` module contains helper functions that take a collection of dynamic processes and wrap them in a more complex dynamic process. Two of such functions are `ofList` and `ofArray`:

Table 4.2: Basic Sequential Functions

Function and type	Description
`Dynamics.ofList: Dynamics<'b> list -> Dynamics<'b list>`	Wraps a list of computations in a compound computation
`Dynamics.ofArray: Dynamics<'b> [] -> Dynamics<'b []>`	Wraps an array of computations in a compound computation

Now we can use one of them to wrap variables `a`, `b` and `c` from our example in a compound dynamic process and then receive data for all these variables. To this list we can also add the built-in `time` variable that returns the current simulation time. This variable is described in section 4.2.

```
> let s = Dynamics.ofArray [| time; a; b; c |];;

val s : Dynamics<float []>

> let r = Dynamics.run s specs |> Seq.toList;;

val r : float [] list =
  [[|0.0; 100.0; 0.0; 0.0|];
   [|1.0; 37.5; 33.33333333; 29.16666667|];
   [|2.0; 14.0625; 25.0; 60.9375|];
   [|3.0; 5.2734375; 14.0625; 80.6640625|];
   [|4.0; 1.977539063; 7.03125; 90.99121094|];
   [|5.0; 0.7415771484; 3.295898438; 95.96252441|];
   [|6.0; 0.2780914307; 1.483154297; 98.23875427|];
   [|7.0; 0.1042842865; 0.6488800049; 99.24683571|];
   [|8.0; 0.03910660744; 0.2780914307; 99.68280196|];
   [|9.0; 0.01466497779; 0.1173198223; 99.8680152|];
   [|10.0; 0.005499366671; 0.0488832593; 99.94561737|]]
```

It is worthy to note that variables `a`, `b` and `c` are simulated in the both cases. If some variable is used in the simulation then it will be simulated whether its data are returned or not with help of the `run` function. This function runs all necessary computations and returns only a specified portion of data. We can return data in any form and use the computation expression syntax in case of need.

For example, the following function is equivalent to the predefined `zip` function from module `Dynamics`.

```
let zip' m1 m2 = dynamics {
```

```
    let! x1 = m1
    let! x2 = m2
    return (x1, x2)
}
```

This function allows us to wrap two computations in one compound computation. The input computations may have different types.

```
> let s2 =
      Dynamics.zip time (Dynamics.zip a (Dynamics.zip b c));;

val s2 : Dynamics<float * (float * (float * float))>

> let r2 = Dynamics.run s2 specs |> Seq.toList;;

val r2 : (float * (float * (float * float))) list =
  [(0.0, (100.0, (0.0, 0.0)));
   (1.0, (37.5, (33.33333333, 29.16666667)));
   (2.0, (14.0625, (25.0, 60.9375)));
   (3.0, (5.2734375, (14.0625, 80.6640625)));
   (4.0, (1.977539063, (7.03125, 90.99121094)));
   (5.0, (0.7415771484, (3.295898438, 95.96252441)));
   (6.0, (0.2780914307, (1.483154297, 98.23875427)));
   (7.0, (0.1042842865, (0.6488800049, 99.24683571)));
   (8.0, (0.03910660744, (0.2780914307, 99.68280196)));
   (9.0, (0.01466497779, (0.1173198223, 99.8680152)));
   (10.0, (0.005499366671, (0.0488832593, 99.94561737)))]
```

The computation, i.e. the simulation, is not started until we explicitly call the `run` function. A value of type `Dynamics<'a>` only returns something (a function) that knows how to simulate the specified dynamic process but the value itself doesn't contain simulation data. Using the built-in variables makes it especially obvious.

4.2 Using Built-in Variables

The `SD` module from the `Maritegra.Aivika` name space has four predefined variables that return the information about the integration nodes. All these variables have type `Dynamics<float>`. Since they are immutable, I will call them values as it is accepted in the functional programming.

Table 4.3: Built-in Variables

Value and type	Description
`starttime:`	Returns the start time

`Dynamics<float>`	
`stoptime:` `Dynamics<float>`	Returns the stop time
`dt:` `Dynamics<float>`	Returns the integration time step
`time:` `Dynamics<float>`	Returns the current simulation time

For illustration we can wrap all four built-in values in one compound computation using the computation expression syntax.

```
> open Maritegra.Aivika;;
> open Maritegra.Aivika.SD;;
> let d1 = dynamics {
      let! x1 = starttime
      let! x2 = stoptime
      let! x3 = dt
      let! x4 = time
      return (x1, x2, x3, x4)
  };;

val d1 : Dynamics<float * float * float * float>
```

Now we can test it using different simulation specs. The computation must always return data corresponded to the specified specs. There is no magic in it. These built-in values are actually implemented as functions that know how to extract the necessary data from the simulation specs.

```
>let makeSpecs x1 x2 x3 = { StartTime=x1;
                            StopTime=x2;
                            DT=x3;
                            Method=RungeKutta4;
                            Randomness=SimpleRnd };;

val makeSpecs : float -> float -> float -> Specs

> makeSpecs 1.0 5.0 1.0 |> Dynamics.run d1 |> Seq.toList;;
val it : (float * float * float * float) list =
  [(1.0, 5.0, 1.0, 1.0); (1.0, 5.0, 1.0, 2.0);
   (1.0, 5.0, 1.0, 3.0); (1.0, 5.0, 1.0, 4.0);
   (1.0, 5.0, 1.0, 5.0)]
```

```
> makeSpecs 10.0 100.0 0.01 |> Dynamics.run d1;;
val it : seq<float * float * float * float> =
  seq
    [(10.0, 100.0, 0.01, 10.0); (10.0, 100.0, 0.01, 10.01);
     (10.0, 100.0, 0.01, 10.02); (10.0, 100.0, 0.01, 10.03); ...]
```

Please note how the fourth item varies. It always contains the current simulation time. Using the `time` built-in value and computation expression syntax, we can create rather complicated dynamic processes varying in time. Two wave functions defined below are a simple example of such processes.

```
let sinWave a p = a * sin (2.0 * Math.PI * time / p)
let cosWave a p = a * cos (2.0 * Math.PI * time / p)
```

These two functions are included in the standard library of Aivika. Like the built-in variables they are defined in the `SD` module.

Table 4.4: The Wave Functions

Function and type	Description
`sinWave: a:Dynamics<float> -> p:Dynamics<float> -> Dynamics<float>`	Returns the sine wave of amplitude `a` and period `p`
`cosWave: a:Dynamics<float> -> p:Dynamics<float> -> Dynamics<float>`	Returns the cosine wave of amplitude `a` and period `p`

To construct computations that would depend on the past, we need additional tools though. Differential equations are an example of such computations. These equations are usually an origin of dynamism in the model. To allow you to define them in an easy and intuitive way close to mathematical notation, Aivika provides a set of built-in functions.

4.3 Declaring Integrals and Differential Equations

The `SD` module contains functions that create integrals as computations of type `Dynamics<float>`. After a simulation is started these computations integrate the underlying differential equations using the specified method and then return the results of integration in the specified nodes. A linear interpolation is used

for all other time values. Thus, the resulting functions look like continuous ones although the integration method returns data in tabular form.

To increase the accuracy of the integration method, you should usually decrease the time step, i.e. increase the number of integration nodes. But it leads to consuming more memory and slowing down the simulation. Therefore it is necessary to find a balance.

Table 4.5: Basic Integral Functions

Function and type	Description

`integ:`
`f:Lazy<Dynamics<float>> ->`
`i:Dynamics<float> ->`
`Dynamics<float>`
	Returns the integral of rate `f` and initial value `i`

`smooth:`
`x:Dynamics<float> ->`
`t:Lazy<Dynamics<float>> ->`
`Dynamics<float>`
	Returns a first order exponential smooth of `x` over time `t`

`smoothI:`
`x:Lazy<Dynamics<float>> ->`
`t:Lazy<Dynamics<float>> ->`
`i:Dynamics<float> ->`
`Dynamics<float>`
	Returns a first order exponential smooth of `x` over time `t` starting at `i`

`smooth3:`
`x:Dynamics<float> ->`
`t:Lazy<Dynamics<float>> ->`
`Dynamics<float>`
	Returns a third order exponential smooth of `x` over time `t`

`smooth3I:`
`x:Lazy<Dynamics<float>> ->`
`t:Lazy<Dynamics<float>> ->`
`i:Dynamics<float> ->`
`Dynamics<float>`
	Returns a third order exponential smooth of `x` over time `t` starting at `i`

`smoothN:`
`x:Dynamics<float> ->`
`t:Lazy<Dynamics<float>> ->`
`n:int ->`
`Dynamics<float>`
	Returns an `n`'th order exponential smooth of `x` over time `t`

`smoothNI:` `x:Lazy<Dynamics<float>> ->`	Returns an `n`'th order exponential smooth of `x` over time `t` starting at `i`

```
t:Lazy<Dynamics<float>> ->
n:int ->
i:Dynamics<float> ->
Dynamics<float>
```

```
delay1:
x:Dynamics<float> ->
t:Dynamics<float> ->
Dynamics<float>
```

Returns a first order exponential delay of `x` for time `t` conserving `x`

```
delay1I:
x:Lazy<Dynamics<float>> ->
t:Dynamics<float> ->
i:Dynamics<float> ->
Dynamics<float>
```

Returns a first order exponential delay of `x` starting with `i` for time `t` conserving `x`

```
delay3:
x:Dynamics<float> ->
t:Dynamics<float> ->
Dynamics<float>
```

Returns a third order exponential delay of `x` for time `t` conserving `x`

```
delay3I:
x:Lazy<Dynamics<float>> ->
t:Dynamics<float> ->
i:Dynamics<float> ->
Dynamics<float>
```

Returns a third order exponential delay of `x` starting with `i` for time `t` conserving `x`

```
delayN:
x:Dynamics<float> ->
t:Dynamics<float> ->
n:int ->
Dynamics<float>
```

Returns an `n`'th order exponential delay of `x` for time `t` conserving `x`

```
delayNI:
x:Lazy<Dynamics<float>> ->
t:Dynamics<float> ->
n:int ->
i:Dynamics<float> ->
Dynamics<float>
```

Returns an `n`'th order exponential delay of `x` starting with `i` for time `t` conserving `x`

The integral functions use the delayed parameters whenever it makes sense. You can consider it as a very useful tool. They actually allow us to define loopbacks explicitly. In most situations the F# compiler itself detects whether the system of differential equations is resolvable or not. Such equations look so natural as they would be written in mathematical notation. The modeler

focuses more on *what* to simulate rather than *how* to simulate. Such a style of defining the task is called *declarative*. For example, a simple model provided in section 4.1 was defined in a declarative manner.

The primary integral function is `integ`. All other functions from this table are derivative and they are expressed through the `integ` function. Below is shown a definition of the `smooth3I` function.

```
let smooth3I x t i =
    let rec y = integ (lazy ((s1 - y) / (t.Value / 3.0))) i
    and s1 = integ (lazy ((s0 - s1) / (t.Value / 3.0))) i
    and s0 = integ (lazy ((x.Value - s0) / (t.Value / 3.0))) i
    in y
```

There are also two auxiliary functions `forecast` and `trend` defined in the `SD` module.

Table 4.6: Auxiliary Integral Functions

Function and type	Description
`forecast: x:Dynamics<float> -> at:Dynamics<float> -> hz:Dynamics<float> -> Dynamics<float>`	Forecasts for `x` over the time horizon `hz` using an average time `at`
`trend: x:Dynamics<float> -> at:Dynamics<float> -> i:Dynamics<float> -> Dynamics<float>`	Returns the fractional change rate of `x` using the average time `at` and initial value `i`

These two functions are defined in Aivika in the following way.

```
let forecast x at hz =
    x * (1.0 + (x / smooth x (lazy at) - 1.0) / at * hz)

let trend x at i =
    (x / smoothI (lazy x) (lazy at) (x / (1.0+i*at)) - 1.0) / at
```

Although the integral functions are constructed in such a way that they allow us to write differential equations declaratively without explicit indicating the order of dependencies between the variables, sometimes it makes sense to introduce an explicit order of relations. Aivika contains class type `Reservoir` for this purpose.

4.4 Defining Reservoirs

The `Reservoir` class type is just a wrapper built on the `integ` function. It has internal fields in which the derivative is memorized. It allows us to refer to the integral value before we define the derivative itself. If F# didn't allow us to define the variables recursively with help of the `rec` keyword then using this class type would be the main way of defining the differential equations.

The following table shows the methods and properties on the `Reservoir` type.

Table 4.7: Methods and Properties on the Reservoir type

Function and type	Description
`new: init:float -> Reservoir`	Creates a new reservoir with the specified initial value
`new: init:Dynamics<float> -> Reservoir`	Creates a new reservoir with the specified initial value
`member Inflow: Dynamics<float>`	Gets and sets the inflow
`member Outflow: Dynamics<float>`	Gets and sets the outflow
`member Value: Dynamics<float>`	Gets the integral value

By default, the `Inflow` and `Outflow` properties return `eta 0.0`. The difference of these properties defines a derivative of the integral:

$$\frac{d}{dt}\texttt{x.Value} = \texttt{x.Inflow} - \texttt{x.Outflow}$$

It is important that we can modify the flow properties at any time even after we made a reference to the integral value somewhere in the equations. It leads us to the following procedure.

1. At first we define constants.
2. At second step we create reservoirs initializing them with the constants.
3. Then we define auxiliary variables that depend on the values of integrals contained in the reservoirs.

4. Finally, we define the flows for the reservoirs, i.e. derivatives.

For instance, we can rewrite the model from section 4.1 using already the reservoirs in accordance with our procedure.

```
> open Maritegra.Aivika
  open Maritegra.Aivika.SD
  ;;
> // 1. Define the constants
  let ka = 1.0
  let kb = 1.0

  // 2. Create reservoirs
  let ra = Reservoir (100.0)
  let rb = Reservoir (0.0)
  let rc = Reservoir (0.0)

  // 3. Define the auxiliaries
  let a = ra.Value
  let b = rb.Value
  let c = rc.Value
  let f = ka * a
  let g = kb * b

  // 4. Define the derivatives
  ra.Outflow <- f
  rb.Inflow <- f
  rb.Outflow <- g
  rc.Inflow <- g
  ;;

val ka : float = 1.0
val kb : float = 1.0
val ra : Reservoir
val rb : Reservoir
val rc : Reservoir
val a : Dynamics<float>
val b : Dynamics<float>
val c : Dynamics<float>
val f : Dynamics<float>
val g : Dynamics<float>
```

It defines the same model but written more imperatively, where we indicate explicitly the dependencies between the variables. The stated above procedure is useful when we cannot put all equations in one, probably huge, `let rec` construct.

The definition of the `Reservoir` class type is very straightforward, although Aivika uses a slightly optimized version.

```
type Reservoir (init: Dynamics<float>) =

    let mutable inflow = eta 0.0
    let mutable outflow = eta 0.0

    let diff = dynamics {

        let! x1 = inflow
        let! x2 = outflow
        return (x1 - x2)
    }

    let value = integ (lazy diff) init

    new (init: double) = Reservoir (eta init)

    member x.Inflow
        with get() = inflow and set (v) = inflow <- v
    member x.Outflow
        with get() = outflow and set (v) = outflow <- v
    member x.Value = value
```

Aivika introduces other class types as well. For example, a `Table` class type is used for working with table functions without which it is impossible to imagine any complex model of System Dynamics.

4.5 Integrating Numerical Functions

The introduced above integration function `integ` can be applied to any time varying numerical function defined somewhere in the ordinary F# code. The idea is very simple. The key point is the `map` function of the `Dynamics` module. This special function allows us to convert the ordinary F# function into a computation of type `Dynamics<float>`. We can take the `time` built-in as the source computation to be transformed.

```
> open Maritegra.Aivika
> open Maritegra.Aivika.SD;;

> let f t = t**2.0 + t + 1.0
> let m = Dynamics.map f time
> let sum = integ (lazy m) (eta 0.0);;

val f : float -> float
```

```
val m : Dynamics<float>
val sum : Dynamics<float>

> let specs =
      { StartTime = 0.0; StopTime = 1.0; DT = 0.001;
        Method = RungeKutta4; Randomness = SimpleRnd };;

val specs : Specs = {StartTime = 0.0;
                     StopTime = 1.0;
                     DT = 0.001;
                     Method = RungeKutta4;
                     Randomness = SimpleRnd;}

> Dynamics.runLast sum specs;;
val it : float = 1.833333333
```

Frankly speaking, the `integ` function does more work than we need here. This function actually returns a dynamic process keeping all its history in the integration nodes under the hood. To get the final sum, we just take the last value.

4.6 Using Table Functions

Given the array of pairs $[|(x_1, y_1); (x_2, y_2); ...; (x_n, y_n)|]$ sorted by x_i, we can create an instance of the `Table` class type and then lookup the specified x using either linear or step-wise interpolation. This class type has the following methods and properties.

Table 4.8: Methods and Properties on the Table type

Function and type	Description

`new:`
`(float * float) [] ->`
`Table`
	Creates a new table using the specified array of pairs $[|(x_1, y_1); ...; (x_n, y_n)|]$ sorted by x_i

`member Lookup:`
`x:Dynamics<float> ->`
`Dynamics<float>`
	Lookups `x` in the table using linear interpolation

`member LookupStepwise:`
`x:Dynamics<float> ->`
`Dynamics<float>`
	Lookups `x` in the table using step-wise interpolation

`member LookupF:` `x:float -> float`	Lookups `x` in the table using linear interpolation

`member LookupStepwiseF: x:float -> float`	Lookups `x` in the table using step-wise interpolation

To simplify the use of tables in the equations that are usually defined in a functional style, the `SD` module of name space `Maritegra.Aivika` defines a set of helper functions.

Table 4.9: Table Functions

Function and type	Description
`table: (float * float) [] -> Table`	Creates a new table using the specified array of pairs $[\|(x_1, y_1); ...; (x_n, y_n)\|]$ sorted by x_i
`lookup: x:Dynamics<float> -> t:Table -> Dynamics<float>`	Lookups `x` in table `t` using linear interpolation
`lookupStepwise: x:Dynamics<float> -> t:Table -> Dynamics<float>`	Lookups `x` in table `t` using step-wise interpolation
`lookupF: x:float -> t:Table -> float`	Lookups `x` in table `t` using linear interpolation
`lookupStepwiseF: x:float -> t:Table -> float`	Lookups `x` in table `t` using step-wise interpolation

Here is a small example of using the table function:

```
let Death_Fraction =
    table [| (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); (1.0, 6.194) |]
    |> lookup (Fish/Carrying_Capacity)
```

We have already a rich enough arsenal of different functions to create rather complicated models of System Dynamics. But in our practice we will need also a tool to define unidirectional flows.

4.7 Representing Unidirectional Flows

A unidirectional flow can be represented as a computation that returns non-negative floating point numbers. The `SD` module contains function `nonnegative` for this purpose. This function is related to other two functions which are equivalents of the standard `max` and `min` functions but for values of type `Dynamics<'a>`.

Table 4.10: Functions maxD, minD and nonnegative

Function and type	Description
`maxD: Dynamics<'a> -> Dynamics<'a> -> Dynamics<'a> when 'a: comparison`	Represents an analog of the standard `max` function
`minD: Dynamics<'a> -> Dynamics<'a> -> Dynamics<'a> when 'a: comparison`	Represents an analog of the standard `min` function
`nonnegative: x:Dynamics<float> -> Dynamics<float>`	Represents an optimized version of expression `maxD x (eta 0.0)`

Perhaps the best way to understand these functions is to look at that how they could be defined using the computation expression syntax. Aivika actually implements them in a more efficient way, working with the computations directly.

```
let maxD m1 m2 = dynamics {
    let! x1 = m1
    let! x2 = m2
    return (max x1 x2)
}

let minD m1 m2 = dynamics {
    let! x1 = m1
    let! x2 = m2
```

```
    return (min x1 x2)
}
```

Then we can apply the `nonnegative` function to return a uniflow. This function is optimized as well. In section 4.8 we'll see an example of using the uniflows.

4.8 Simulating Fish Bank Model

The Fish Bank model is distributed along with other sample models as a part of the installation package of Simtegra MapSys. 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. The model diagram is shown on figure 4.1.

Perhaps the best way to define the mathematical equations for this model is to provide their equivalent in Aivika but written with help of the recursive `let` construct. It has a striking likeness to that how the same model is defined in MapSys, which specialized language is more high level.

4.8.1 Approach Number 1. Declaring Model Equations

We can try to define the equations declaratively without explicit indicating the order of dependencies[4], which is invaluable for working with large models. This approach works, because all loopbacks are defined lazily. This is why the integral functions require delayed values for the arguments. What is important is that the F# compiler itself detects whether the system of equations can be integrated or cannot because of existence of circular relationships.

```
open Maritegra.Aivika
open Maritegra.Aivika.SD

let rec Annual_Profit = Profit
and Area = 100.0;
and Carrying_Capacity = 1000.0
and catch_per_Ship =
    table [| (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) |]
    |> lookup Density
and Death_Fraction =
```

[4]Unfortunately, at time of writing this book the recent version (1.9.9.9) of the F# compiler generates an incorrect code for this model, which leads to raising the `NullReferenceException` exception during execution. The author hopes that this bug about which the F# team is well-informed will be fixed soon. In case of need you can always try another approach described in section 4.8.2

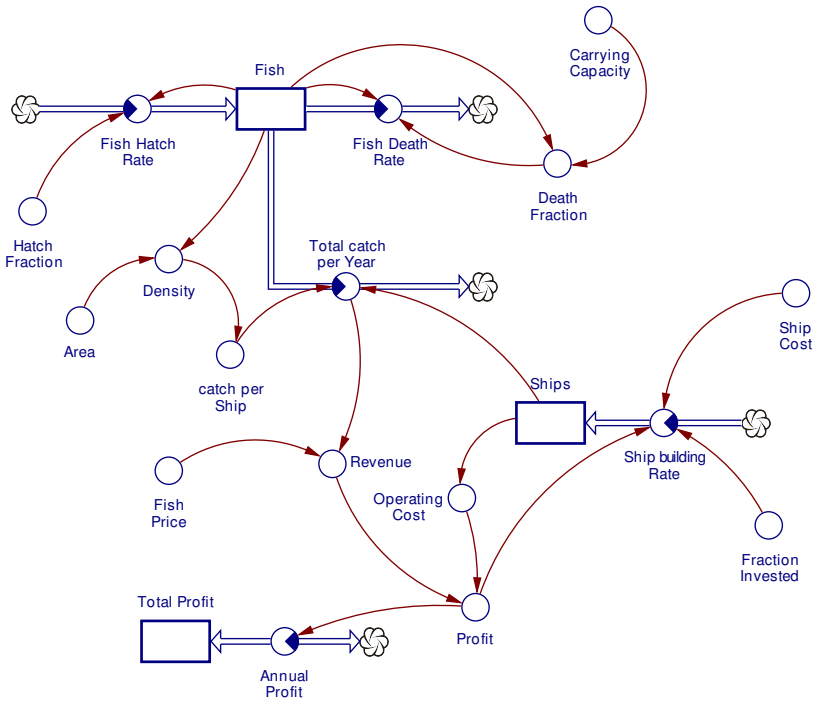

Figure 4.1: Fish Bank Model

```
    table [| (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); (1.0, 6.194) |]
    |> lookup (Fish/Carrying_Capacity)
and Density = Fish/Area
and Fish = integ (lazy (Fish_Hatch_Rate - Fish_Death_Rate
                                        - Total_catch_per_Year))
                 (eta 1000.0)
and Fish_Death_Rate = nonnegative (Fish*Death_Fraction)
and Fish_Hatch_Rate = nonnegative (Fish*Hatch_Fraction)
and Fish_Price = 20.0
and Fraction_Invested = 0.2
and Hatch_Fraction = 6.0
and Operating_Cost = Ships*250.0
and Profit = Revenue-Operating_Cost
and Revenue = Total_catch_per_Year*Fish_Price
and Ship_building_Rate =
    nonnegative (Profit*Fraction_Invested/Ship_Cost)
and Ship_Cost = 300.0
and Ships = integ (lazy Ship_building_Rate) (eta 10.0)
and Total_catch_per_Year = nonnegative (Ships*catch_per_Ship)
and Total_Profit = integ (lazy Annual_Profit) (eta 0.0)
```

Please pay attention to that how the stocks are defined with help of the `integ` function and that how the unidirectional flows are defined with help of function `nonnegative`. The model also defines two table functions.

Now we can simulate the whole model and return results for each of the variables. Let's suppose that we are interested in the amount of fish, the number of ships and the value of the annual profit. To interpret the results, we also need to know the simulation time. The corresponded compound dynamic process can be defined in the following way:

```
let s = Dynamics.ofArray [| time; Fish; Annual_Profit; Ships |]
```

Now we can run the simulation for any specified specs and receive the results. But before it I'll show how the same model can be defined with help of reservoirs.

4.8.2 Approach Number 2. Ordering Model Equations

The reservoirs described in section 4.4, page 26, allows us to write differential equations in a few steps without explicit using recursion. The drawback of this approach is that we have to place the equations in right order. Now they can be rewritten in the following way for our example.

```
open Maritegra.Aivika
open Maritegra.Aivika.SD
```

```
// 1. Define constants

let Area = 100.0;
let Carrying_Capacity = 1000.0
let Fish_Price = 20.0
let Fraction_Invested = 0.2
let Hatch_Fraction = 6.0
let Ship_Cost = 300.0

// 2. Create reservoirs

let Fish_R = Reservoir (1000.0)
let Ships_R = Reservoir (10.0)
let Total_Profit_R = Reservoir (0.0)

let Fish = Fish_R.Value
let Ships = Ships_R.Value
let Total_Profit = Total_Profit_R.Value

// 3. Define auxiliaries

let Density = Fish/Area
let catch_per_Ship =
    table [| (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) |]
    |> lookup Density
let Death_Fraction =
    table [| (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); (1.0, 6.194) |]
    |> lookup (Fish/Carrying_Capacity)
let Fish_Death_Rate = nonnegative (Fish*Death_Fraction)
let Fish_Hatch_Rate = nonnegative (Fish*Hatch_Fraction)
let Operating_Cost = Ships*250.0
let Total_catch_per_Year = nonnegative (Ships*catch_per_Ship)
let Revenue = Total_catch_per_Year*Fish_Price
let Profit = Revenue-Operating_Cost
let Annual_Profit = Profit
let Ship_building_Rate =
    nonnegative (Profit*Fraction_Invested/Ship_Cost)

// 4. Define derivatives
```

```
Fish_R.Inflow <-
    Fish_Hatch_Rate - Fish_Death_Rate - Total_catch_per_Year
Ships_R.Inflow <- Ship_building_Rate
Total_Profit_R.Inflow <- Annual_Profit
```

Also we'll define the same output variables.

```
let s = Dynamics.ofArray [| time; Fish; Annual_Profit; Ships |]
```

Finally, we can proceed to the simulation and receive a sequence of output arrays for the specified variables.

4.8.3 Receiving Results of Simulation

Let's specify the same specs as in the model distributed together with Simtegra MapSys. The start time will be 0.0, stop time will be 13.0, the integration time step will be 0.02. We'll apply the 4th order Runge-Kutta method and select a simple pseudo-random generator. We have to define the latter even if we are not going to use random functions.

```
let specs = {

    StartTime=0.0; StopTime=13.0; DT=0.02;
    Method=RungeKutta4; Randomness=SimpleRnd
}
```

Now if we'll try to start the simulation then we'll receive a reply almost immediately. Here Aivika returns a sequence of values, i.e. an enumeration of the values which are yet to be calculated.

```
> let results = Dynamics.run s specs;;

val results : seq<float []>
```

It means that the simulation is actually not started. To run it finally, we have to start enumerating the elements of this sequence. The simulation can allocate a memory for storing its intermediate results. This memory will be released after we request the last element of the sequence. If we'll try to enumerate the elements of the same sequence again, then a new simulation will be run in response. In this example we don't use random functions, which are described in section 4.9, page 38. Therefore we'll always receive the same results for each new simulation. But if we used them then the results could be different. In other words each new enumeration of the sequence with results of the simulation may return different data.

```
> results |> Seq.take 10 |> Seq.toList;;
val it : float [] list =
  [[|0.0; 1000.0; 2504.333333; 10.0|];
   [|0.02; 991.1654131; 2506.509107; 10.03340558|];
   [|0.04; 982.4285895; 2508.719677; 10.0668404|];
   [|0.06; 973.7869922; 2510.963999; 10.10030492|];
   [|0.08; 965.2381583; 2513.241053; 10.13379958|];
   [|0.1; 956.779697; 2515.544385; 10.16732481|];
   [|0.12; 948.4092957; 2517.869661; 10.20088087|];
   [|0.14; 940.1247; 2520.22477; 10.23446813|];
   [|0.16; 931.9237158; 2522.608781; 10.26808699|];
   [|0.18; 923.8042107; 2525.020788; 10.30173782|]]
```

Actually, such a sequence is not a single way or receiving the results. Using the computation expression syntax, you can create computations of any complexity. It also means that you can save the results of the simulation using these expressions. Section 8.3, page 95, covers this topic in more detail. But you should remember that such a technique is not safe and that it is error-prone especially if you are going to run parallel simulations as described in section 8.1, page 95. The most safe and robust way is to pass the results directly through the `run` function or their counterparts `runLast` and `runWhile`, although it may have some overheads.

4.8.4 Saving Results in CSV File

The results can be saved in the CSV file to analyze. The following utility function can be helpful here.

```
open System
open System.IO
open System.Text

let saveCSV (results: #seq<float []>)
            (fields: (string * int) list)
            (filename: string) =

    let encoding = UTF8Encoding (true) :> Encoding

    use stream = new FileStream (filename, FileMode.Create)
    use file = new StreamWriter (stream, encoding)

    let s = fields
            |> List.map (fun (name, _) -> name)
            |> List.reduce (fun s1 s2 -> s1 + "\t" + s2)

    file.WriteLine (s)
```

```
    for r in results do

        let s = fields
                |> List.map (fun (_, id) -> sprintf "%f" r.[id])
                |> List.reduce (fun s1 s2 -> s1 + "\t" + s2)

        file.WriteLine (s)
```

This small function takes the results, a schema that describes the fields we desire to see in the file and the file name. The schema is a list of pairs consisting of the field name and its index in the resulting array. The function traverses the results and save those of them which were indicated in the schema.

For example, to save the output data in the *FishBank.csv* file, we can enter in the F# Interactive:

```
> let fs = [("time", 0); ("Fish", 1); ("Annual_Profit", 2);
            ("Ships", 3)];;

val fs : (string * int) list =
  [("time", 0); ("Fish", 1); ("Annual_Profit", 2); ("Ships", 3)]

> saveCSV results fs "FishBank.csv";;
val it : unit = ()
```

This model was deterministic but Aivika can simulate stochastic models as well. Please read the next section to know how you can use the built-in random functions and create your own.

4.9 Using Random Functions

Aivika provides a set of the predefined random functions. So, there are generators for the uniform, normal, binomial and Poisson random values. Like many other software tools specialized in System Dynamics Aivika represents basic random functions as discrete processes that return new random numbers at each iteration step, i.e. as piecewise constant functions varying in the primary integration nodes. You can find more information about the discrete processes in section 4.10, page 42.

The built-in random functions are collected in the table below.

Table 4.11: Random Functions

Function and type	Description
`random: a:Dynamics<float> ->`	Returns the uniform random numbers between `a` and `b`

`b:Dynamics<float> ->`
`Dynamics<float>`

`randomS:` Returns the uniform random numbers between
`s:int ->` `a` and `b` with constant seed `s`
`a:Dynamics<float> ->`
`b:Dynamics<float> ->`
`Dynamics<float>`

`normal:` Returns the normal random numbers with
μ`:Dynamics<float> ->` mean μ and variance ν
ν`:Dynamics<float> ->`
`Dynamics<float>`

`normalS:` Returns the normal random numbers with
`s:int ->` mean μ, variance ν and constant seed `s`
μ`:Dynamics<float> ->`
ν`:Dynamics<float> ->`
`Dynamics<float>`

`binomial:` Returns the binomial random numbers on
`p:Dynamics<float> ->` `n` trials of probability `p`
`n:Dynamics<int> ->`
`Dynamics<int>`

`binomialS:` Returns the binomial random numbers on
`s:int ->` `n` trials of probability `p` with constant
`p:Dynamics<float> ->` seed `s`
`n:Dynamics<int> ->`
`Dynamics<int>`

`poisson:` Returns the Poisson random numbers with
μ`:Dynamics<float> ->` mean μ
`Dynamics<int>`

`poissonS:` Returns the Poisson random numbers with
`s:int ->` mean μ and constant seed `s`
μ`:Dynamics<float> ->`
`Dynamics<int>`

`exprnd:` Returns the exponential random numbers
μ`:Dynamics<float> ->` with mean μ
`Dynamics<int>`

`exprndS:` Returns the exponential random numbers
`s:int ->` with mean μ and constant seed `s`

μ:Dynamics<float> -> Dynamics<int>

Each function has its counterpart with suffix `S` in the name. Such a counterpart takes a constant seed as the first parameter. To replicate the stream of random numbers, you should define a non-zero integer seed. Also some arguments are computations of type `Dynamics`. It means that they can vary in time. It allows us to build new computations with rather sophisticated behavior.

If we had no function `exprnd` then we could define it in the following way:

```
let exprnd (mu: Dynamics<float>) =
    - log (random (eta 0.0) (eta 1.0)) * mu
```

There is a special function `randomizer` with help of which we can also create new random functions. It uses type `Generator` representing any function that possibly creates some side-effect and then returns a floating point number.

```
type Generator = unit -> float
```

The function signature and description are provided below. All the stated above random functions are actually implemented based on this function. The corresponded definitions are simple enough. You can also create your own random functions this way.

Table 4.12: Randomizer Function

Function and type	Description
randomizer: s:int option -> tr:(Generator -> Generator) -> m:Dynamics<Generator -> 'a> -> Dynamics<'a>	Returns random numbers using the specified seed `s`, generator transform `tr` and generating process `m`

Aivika initially creates the uniform random number generator that returns floating point numbers between `0.0` and `1.0`. This generator is transformed by parameter `tr`. It happens before the simulation is started. Then during a computation this transformed generator is passed to process `m` that already returns random numbers which become a result. If the constant seed is not specified than we should pass `None` to parameter `s`.

For example, the built-in `randomS` is defined in the following way.

```
let randomS s (a: Dynamics<float>) (b: Dynamics<float>) =
    dynamics {
        let! a' = a
```

```
        let! b' = b
        return (fun g -> a' + (b' - a') * g ())
    } |> randomizer (Some s) id
```

To implement the `normalS` function, we have to define the generator transform first. It takes the uniform random number generator and returns the normal random number generator with mean `0.0` and variance `1.0`.

```
let normalGenerator (g: Generator) =

    let next = ref 0.0
    let flag = ref false

    in fun () ->

        if !flag then

            flag := false
            !next

        else

            let mutable xi1 = 0.0
            let mutable xi2 = 0.0
            let mutable psi = 0.0

            while (psi >= 1.0) || (psi = 0.0) do

                xi1 <- 2.0 * g () - 1.0
                xi2 <- 2.0 * g () - 1.0
                psi <- xi1 * xi1 + xi2 * xi2

            psi <- sqrt (- 2.0 * (log psi) / psi)

            flag := true
            next := xi2 * psi

            xi1 * psi
```

Then the random function itself can be defined like this:

```
let normalS s (mu: Dynamics<float>) (sigma: Dynamics<float>) =
    dynamics {
        let! mu' = mu
        let! sigma' = sigma
        return (fun g -> mu' + sigma' * g ())
    } |> randomizer (Some s) normalGenerator
```

The random functions are not only discrete functions defined in the standard library of Aivika. Some of them are described in the next sections as well.

4.10 Introducing Discrete Processes and Functions

In Aivika the *discrete* functions are a particular case of more general discrete processes that may return any values during the simulation, not only numbers. In its turn the discrete process is such a computation that returns a value that doesn't change except of the integration intervals regardless of the integration method used.

4.10.1 Creating Discrete Processes

There are two very important functions that can be applied to any computation in the Dynamics workflow, that is a value of generic type `Dynamics<'a>`. The both functions return discrete processes. The functions are defined in two modules `SD` and `Dynamics`.

Table 4.13: Basic Discrete Functions

Function and type	Description

`init:`
`Dynamics<'a> ->`
`Dynamics<'a>`
	Returns the initial value

`discrete:`
`Dynamics<'a> ->`
`Dynamics<'a>`
	Returns a discrete estimate

The first function is useful if we need to know the initial value of the specified computation. Then only one value is returned for all integration intervals. Therefore the resulting computation is a discrete process according to the definition.

The second function is a more subtle thing. If the input computation isn't discrete then the resulting one is a discrete estimate for the former. It computes values of the input computation exactly in the integration nodes and then extends these values to the corresponded integration intervals, which makes obviously the result discrete.

In fact, we can request for the result of computation at any time t and this time can differ from the integration nodes. Moreover, the Runge-Kutta method complicates the matter. All this should be taken into account. What you must know is that the `discrete` function is very cheap and it returns a

rather accurate discrete estimate for the input computation. The function is efficient and it is often applied in Aivika itself. Also you can combine this function with memoization to order the calculations creating strictly sequential processes. You can find more details in section 8.2, page 95.

4.10.2 Miscellaneous Discrete Functions

Like other simulation tools Aivika provides the pulse and step functions. These functions are discrete.

Table 4.14: Miscellaneous Discrete Functions

Function and type	Description

`step:`
`h:Dynamics<float> ->`
`st:Dynamics<float> ->`
`Dynamics<float>`
	Returns 0 until time `st` and then returns `h`

`pulse:`
`st:Dynamics<float> ->`
`w:Dynamics<float> ->`
`Dynamics<float>`
	Returns the pulse of height 1 starting at time `st` with duration `w`

`pulseP:`
`st:Dynamics<float> ->`
`w:Dynamics<float> ->`
`p:Dynamics<float> ->`
`Dynamics<float>`
	Returns the pulse of height 1 starting at time `st` with duration `w` and period `p`

`ramp:`
`slope:Dynamics<float> ->`
`st:Dynamics<float> ->`
`e:Dynamics<float> ->`
`Dynamics<float>`
	Returns 0 until time `st` and then slopes until time `e` and then holds `slope`

These functions can be easily defined using the computation expression syntax. For example, the `step` function is equivalent to the following one.

```
let step h st =
    dynamics {
        let! st' = st
        let! t'  = time
        let! dt' = dt
        if st' < t' + dt' / 2.0 then
```

```
            return! h
        else
            return 0.0
    } |> discrete
```

Please note how the result becomes discrete. In a similar way you can define your own discrete functions. You can define something using the `dynamics` builder and then pass the result to function `discrete` that already creates a discrete process.

4.10.3 Delaying Computations

The discrete processes have one excellent feature. They can be memoized during the simulation. It allows us to refer to their past values. Aivika contains two useful functions that delay the computations. Only we should remember that the result is discrete in the both cases.

Table 4.15: Delay Functions

Function and type	Description

`delay:`
`x:Dynamics<'a> ->`
`d:Dynamics<float> ->`
`Dynamics<'a>`
	Returns a delayed discrete value of `x` using a lag time of `d`

`delayI:`
`x:Lazy<Dynamics<'a>> ->`
`d:Dynamics<float> ->`
`i:Dynamics<'a> ->`
`Dynamics<'a>`
	Returns a delayed discrete value of `x` using a lag time of `d` and initial value `i`

Please note that the both functions are generic, that is they will work with any computation in the Dynamics workflow. Also the first argument of the second function was intentionally made delayed, which allows us to use this function in the recursive `let` construct as it was in case of the integrals.

```
let rec fibs =
    delayI (lazy fibs) (2.0 * dt) (eta 0.0) +
    delayI (lazy fibs) dt (eta 1.0)
```

The delay functions can also be modeled with help of more heavy-weight conveyors which are described in the next section.

4.11 Using Discrete Stocks

4.11.1 Using Conveyors

4.11.2 Using Ovens

4.11.3 Using Queues

4.12 Working with Arrays

TODO: Estimating Mean Value and Variance.
TODO: An example with `smoothNI`.

4.13 Calling External Functions within Simulation

An ability to call external functions within the simulation is inherited in workflow `Dynamics`. We can use the computation expression syntax for this purpose. But we must understand that the computation returns multiple values and the order in which these values are calculated and time at which it happens is undefined in general case.

4.13.1 Calling Pure Functions

If your function is *pure*, i.e. has no assignment nor any other side-effect, then there is no problem. The result of such a function depends on nothing but the arguments values.

```
open Maritegra.Aivika
open Maritegra.Aivika.SD

let rho (x1, y1) (x2, y2) =
    let dx: float = x2 - x1
    let dy: float = y2 - y1
    in sqrt (dx*dx - dy*dy)

let rhoD m1 m2 = dynamics {
    let! p1 = m1
    let! p2 = m2
    return (rho p1 p2)
}
```

In this example function `rho` is pure. Therefore we can use it within any computation expression. The result will always be predictable in whatever simulation we will use this function.

4.13.2 Sequencing Function Calls

It must be admitted that many practical functions are not pure. Fortunately, we can use such functions within the simulation too. The `Dynamics` module contains three helper functions `memo`, `memo0` and `memoGenerator` that introduce a sequential order of calculations for the specified computation of type `Dynamics`. These functions are described in section 8.2, page 95, of this book.

The following example returns a computation of Fibonacci numbers:

```
// a function with side-effect!
let fib =
    let a = ref 0
    let b = ref 1
    in fun () ->
        let va = !a
        let vb = !b
        a := vb
        b := va + vb
        va

let fibD =
    dynamics {
        return (fib ())
    }
    |> Dynamics.memo0 discrete
```

Here the generator of numbers, function `fib`, has a side-effect. But dynamic process `fibD` always returns numbers sequentially. The applied `memo0` function gives a guarantee that the calculations will be called one by one for the integration nodes in a strongly sequential order[5]. But if we'll try to run the simulation two times then we'll receive different results.

```
> let specs = { StartTime=0.0;
                StopTime=0.2;
                DT=0.01;
                Method=RungeKutta4;
                Randomness=SimpleRnd };;

val specs : Specs = {StartTime = 0.0;
                     StopTime = 0.2;
                     DT = 0.01;
                     Method = RungeKutta4;
                     Randomness = SimpleRnd;}

> Dynamics.run fibD specs |> Seq.toList;;
```

[5]This rule works even if the `run` function is replaced with `runLast`.

```
val it : int list =
  [0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; 144; 233;
   377; 610; 987; 1597; 2584; 4181; 6765]

> Dynamics.run fibD specs |> Seq.toList;;
val it : int list =
  [10946; 17711; 28657; 46368; 75025; 121393; 196418;
   317811; 514229; 832040; 1346269; 2178309; 3524578;
   5702887; 9227465; 14930352; 24157817; 39088169;
   63245986; 102334155; 165580141]
```

To force the `fibD` process to return the same results, we can reset the generator at the initial time of simulation. In this case using a class type can be a better choice.

```
type Fib2 () =
    let mutable a = 0
    let mutable b = 1
    member x.Reset () =
        a <- 0
        b <- 1
    member x.Next () =
        let va = a
        let vb = b
        a <- vb
        b <- va + vb
        va

let fib2D =
    let g = Fib2 ()
    dynamics {
        let! t0 = starttime
        let! t  = time
        if t = t0 then
            g.Reset ()
        return (g.Next ())
    }
    |> Dynamics.memo0 discrete
```

Now we see that the modified version of the dynamic process returns numbers starting from zero for the second simulation too.

```
> Dynamics.run fib2D specs |> Seq.toList;;
val it : int list =
  [0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; 144; 233;
   377; 610; 987; 1597; 2584; 4181; 6765]
```

```
> Dynamics.run fib2D specs |> Seq.toList;;
val it : int list =
  [0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; 144; 233;
   377; 610; 987; 1597; 2584; 4181; 6765]
```

This approach is sufficient in many situations, although it has one weakness. It would not be thread-safe to use the `fib2D` computation if a few simulations were launched simultaneously, for different simulations would modify one shared variable `g`. As we can see, impure functions may cause so many problems. But even for this case there is a solution. So, using the DynamicsCont workflow described further, we can create a discrete process that will generate safely the Fibonacci numbers. Please see section 6.8, page 73, for more details.

Chapter 5

DynamicsCont Workflow Basics

In addition to the `Dynamics` workflow Aivika has also the `DynamicsCont` workflow. The latter is a flexible bridge between the process-oriented DES and other modeling paradigms supported by Aivika. In many senses the `DynamicsCont` workflow is similar to `Dynamics`. Each of them means some delayed computation, which is started during a simulation. The both workflows allow you to write complicated F# code using the computation expression syntax and define rather sophisticated computations. Also you can call external .NET functions within your computations, the feature which is difficult to overestimate.

Nevertheless, there is an important difference between these two workflows. A computation in the `DynamicsCont` workflow is like an operating system process or thread. Its control flow can be suspended at any time and then resumed later. This is a key point for the process-oriented simulation.

What allows us to connect the process-oriented simulation to other parts of the hybrid model, for example, written with help of differential equations, comes from the functional programming. I will describe it in detail later in section 9, page 97. What you must know now is that any computation in the `Dynamics` workflow can be a part of computation in the `DynamicsCont` workflow. For example, it allows us to use any variables and functions of System Dynamics in the process-oriented simulation. In the functional programming this mechanism is called *lifting*. At the same time there is a connection in opposite direction. We can convert the process-oriented simulation into a computation in the `Dynamics` workflow. All this allows us to truly create hybrid models.

5.1 Using Computation Expression Syntax

The `DynamicsCont` workflow builder is called `dynamicscont`. It supports all keywords that the builder can. We can put almost any ordinary F# code inside of a computation expression as it was in case of the `Dynamics` workflow. If you

worked with the latter or the standard `async` workflow then you will be able to use the `dynamicscont` workflow as well. The main idea is the same.

```
let next i = dynamicscont {
    printfn "i = %i" i
    return (i + 1)
}

let count n = dynamicscont {
    let i = ref 1
    while !i <= n do
        let! j = next !i
        i := j
}
```

5.2 Lifting Computation

To build hybrid models, we must know how to combine computations in the `Dynamics` workflow with computations in the `DynamicsCont` workflow. The lift operation allows us to transform more simple computation into more complex. Here the `Dynamics` workflow is more simple which the `DynamicsCont` workflow is based on. The corresponded lift function is defined in the `DynamicsCont` module.

Table 5.1: Lift Function

Function and type	Description
`DynamicsCont.lift: Dynamics<'a> -> DynamicsCont<'a>`	Lifts the computation

I will often use this function in the process-oriented models. For example, to know the current simulation time, I have to lift the `time` built-in. Similarly, to use the current value of integral `A` in the the process-oriented simulation, I can apply the `lift` function to `A`. This function is indeed very useful.

5.3 Running Computation

The *run* function is important for every computation. This is what starts the computation and then allows us to get its result. The `Dynamics` workflow is more simple and its `run` function starts the computation immediately. The `DynamicsCont` workflow is not that case. Its `run` function reduces a computation in the `DynamicsCont` workflow to some computation in the `Dynamics` workflow. Also the function takes as an argument another function that will process the

result of the source computation.

Table 5.2: Run Function

Function and type	Description
`DynamicsCont.run: DynamicsCont<'a> -> f:('a -> unit) -> Dynamics<unit>`	Runs the computation which result will be processed by function `f`

Aivika is built in such a way that in most cases you won't need to use this `run` function directly. Also you can notice that this function is somewhere similar to the `lift` function. Indeed, you can think of the `run` function as an up-cast conversion, while the `lift` function resembles slightly the down-cast conversion for the class type hierarchy, although this analogy is very weak. The run and lift functions are traits of the functional programming which is quite different from the object-oriented programming. If you are familiar with the functional programming then you might notice that the `DynamicsCont` workflow is a monad transformer parameterized by the `Dynamics` monad. But to use the both workflows successfully in your simulation model, it is unnecessary to be an expert in the functional programming.

Chapter 6

Discrete Event Simulation (DES)

In chapter 4 we considered functions and methods that allow us to simulate models of System Dynamics (SD). Such models mainly describe *continuous* variables, although there are some exceptions. For example, the SD model may contain a *discrete* conveyor that changes incrementally. Discrete Event Simulation (DES) involves simulating such variables that change in discrete steps. Then an event implies some variable change.

Aivika supports three main paradigms of DES. You can create *activity-oriented*, *event-oriented* and *process-oriented* models. Moreover, you can combine all these three paradigms together with System Dynamics in one hybrid model including the agent-based modeling described further in chapter 7. The key to success consists in using two workflows `Dynamics` and `DynamicsCont`.

6.1 Applying Activity-oriented Paradigm

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*.

The time points are integration nodes in our case. To simulate an activity, we'll create a computation in the `Dynamics` workflow. Using the computation expression syntax, we can define a rather complicated behavior. To get the current simulation time, we'll apply the `let!` construct inside of the computation. The code of the computation itself will be executed sequentially for each time point, i.e. the integration node.

I'll illustrate the approach on the following sample model [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.

Here is one of possible solutions.

```
open System

open Maritegra.Aivika
open Maritegra.Aivika.SD

let specs = {

    StartTime=0.0; StopTime=10000.0; DT=0.05;
    Method=RungeKutta4; Randomness=SimpleRnd
}

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

let mutable totalUpTime = 0.0   // total up time for all machines

let expovariate =
    let rnd = new Random ()
    in fun lambda ->
        - log (rnd.NextDouble ()) / lambda

let machine id =

    let upTimeNum = ref -1
    let repairTimeNum = ref -1
    let startUpTime = ref 0.0

    dynamics {

        if !upTimeNum > 0 then
            decr upTimeNum

        if !upTimeNum = 0 then
            upTimeNum := -1

            // the machine is broken
            let! finishUpTime = time
            let! dt' = dt
            totalUpTime <- totalUpTime
                + (finishUpTime - !startUpTime)
```

```
            repairTimeNum := int <|
                (expovariate repairRate) / dt'

        elif !upTimeNum < 0 then

            if !repairTimeNum > 0 then
                decr repairTimeNum

            if !repairTimeNum = 0 then
                repairTimeNum := -1

                // the machine is repaired
                let! t' = time
                let! dt' = dt
                startUpTime := t'
                upTimeNum := int <| (expovariate upRate) / dt'

            elif !repairTimeNum < 0 then

                // initialization
                let! t' = time
                let! dt' = dt
                startUpTime := t'
                upTimeNum := int <| (expovariate upRate) / dt'

    } |> Dynamics.memo0 discrete

let machine1 = machine 1
let machine2 = machine 2

let system = dynamics {

    do! machine1    // process machine 1
    do! machine2    // process machine 2
    return! (totalUpTime / (2.0 * stoptime))
}

[<EntryPoint>]
let main args =

    let result = Dynamics.runLast system specs

    printfn "Long-run proportion of up time = %f" result

    0
```

The mean up time is 1.0 and the mean repair time is 0.5. Therefore it is sufficient to take DT = 0.05. We have to generate such values for the up time and repair time so that they will be multiple of DT. Here we use counters of iterations that should pass before the machine is switched to another state. The counters have suffix Num in the name.

```
let! dt' = dt
repairTimeNum := int <| (expovariate repairRate) / dt'
```

To get DT, we apply the let! construct to built-in computation dt that knows how to calculate DT defined in the simulation specs.

```
let specs = {

    StartTime=0.0; StopTime=10000.0; DT=0.05;
    Method=RungeKutta4; Randomness=SimpleRnd
}
```

In general, the order of calculations is not defined for the computation in the Dynamics workflow. Now we want to create a dynamic process (in mathematical sense) that calculates values () of type unit in the time points and also generates some side-effect. We are mostly interested in this side-effect that leads to an update of the totalUpTime variable. We need such calculations that would be performed strongly sequentially in the time points. Here we apply the memo0 function described further in section 8.2, page 95.

```
dynamics {

    // thanks to the memo0 function,
    // this code we'll be executed sequentially
    // in the time points, i.e. the integration nodes
    ...

} |> Dynamics.memo0 discrete
```

Then we create two machines machine1 and machine2, each of them is a computation of type Dynamics<unit>. To involve simulating the both machines, in the main model we apply the do! construct that binds computations.

```
let system = dynamics {

    do! machine1    // process machine 1
    do! machine2    // process machine 2
    return! (totalUpTime / (2.0 * stoptime))
}
```

The rest part of the code starts a simulation and then outputs the result. We are interested only in the last value. Please note that without function memo0 the machines wouldn't actually be simulated. Aivika would launch their computations only once for the last time point.

```
let result = Dynamics.runLast system specs
printfn "Long-run proportion of up time = %f" result
```

You might notice that this approach of simulating two machines is not efficient at all. There are too many time-wasting iterations which we have to traverse waiting for the occurrence of events and decreasing our counters `upTimeNum` and `repairTimeNum`. Fortunately, there are more efficient approaches. Two of them are described further. You will see how the same task can be solved in significantly more efficient ways.

6.2 Using Event Queue

All other simulation paradigms described further in the book, including the agent-based modeling, are implemented on top of the event queue represented in Aivika by class type `Env`. Internally, `Env` is based on an efficient heap-based *priority queue*. An object value of this class type allows us to dispatch events, which are activated then at the specified time.

It may seem ironical but the events are just computations in the `Dynamics` workflow. If you want to transfer some data with the event then you can use a *closure*. It is so natural for the functional programming language. You will see an example in section 6.3, page 58.

The `Env` class type has the following methods.

Table 6.1: Methods on the Env type

Function and type	Description
`new:` `unit -> Env`	Creates a new event queue

`member Dispatch:`
`t:Dynamics<float> *`
`c:Dynamics<unit> ->`
`Dynamics<unit>`
	Activates event `c` at time `t`

`member DispatchD:`
`t:Dynamics<float> *`
`c:Dynamics<unit -> unit> ->`
`Dynamics<unit>`
	Activates event `c` at time `t`

`member Run:` `Dynamics<unit>`	Runs the event queue

The dispatching methods activate the events at the specified time. It hap-

pens only if the event queue itself is involved in the simulation. To involve, we have to use the `Run` method and bind its result with the main computation like this.

```
let e = Env ()

let system = dynamics {
    ...
    do! e.Run    // run the event queue
    ...
}
```

The events have either type `Dynamics<unit>` or `Dynamics<unit -> unit>`. The latter is needed for the `DynamicsCont` workflow with help of which the process-oriented paradigm is implemented in Aivika. Every event is a computation. When its activation time comes, event's computation is involved in the main simulation. It provides us with fantastic facilities of building hybrid models.

The activation time can be any except for one requirement. The final value must be greater than or equaled to the current simulation time. In other words, the event queue cannot dispatch events back to the past. But the time value can differ from the integration nodes defined by such methods as Runge-Kutta. The time value can be arbitrary. It allows us to build exact simulation models. What is important is that all this is well integrated with module of System Dynamics. The `Dynamics` workflow performs all the hard work.

The activation time is also a computation of type `Dynamics<float>`. It was done for generality and for easy using built-ins `time`, `dt` and `starttime`. If you want to pass the specified time value then you can always use the `eta` function that returns a new computation.

The both dispatching methods `Dispatch` and `DispatchD` return computations of type `Dynamics<unit>`. You must involve these computations in the simulation using the `do!` construct as it will be shown in the next section. which demonstrates how you can work directly with these methods of the event queue.

6.3 Applying Event-oriented Paradigm

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*.

To apply the scheme, we use the `Env` class type that implements the event queue. I'll illustrate the approach for the same sample model which was considered in section 6.1, page 53. Please pay attention to that how I transfer the event data through the closure.

```
#nowarn "40"

open System

open Maritegra.Aivika
open Maritegra.Aivika.SD

let specs = {

    StartTime=0.0; StopTime=10000.0; DT=0.05;
    Method=RungeKutta4; Randomness=SimpleRnd
}

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

let mutable totalUpTime = 0.0   // total up time for all machines

let expovariate =
    let rnd = new Random ()
    in fun lambda ->
        - log (rnd.NextDouble ()) / lambda

let machine id (e: Env) =

    let rec broken startUpTime =
        dynamics {

            // the machine is broken
            let! finishUpTime = time
            totalUpTime <- totalUpTime
                + (finishUpTime - startUpTime)
            let repairTime = expovariate repairRate

            // register a new event
            do! e.Dispatch (eta (finishUpTime + repairTime),
                    repaired)
        }
    and repaired =
        dynamics {

            // the machine is repaired
            let! startUpTime = time
            let upTime = expovariate upRate

            // register a new event
```

```
            do! e.Dispatch (eta (startUpTime + upTime),
                    broken startUpTime)    // closure
        }
    and started =
        e.Dispatch (starttime, repaired)

    in Dynamics.once started

let e = Env ()

let machine1 = machine 1 e
let machine2 = machine 2 e

let starter =
    dynamics {
        do! machine1
        do! machine2
    } |> Dynamics.once

let system = dynamics {

    do! starter     // start the machines
    do! e.Run       // run the event queue

    return! (totalUpTime / (2.0 * stoptime))
}

[<EntryPoint>]
let main args =

    let result = Dynamics.runLast system specs

    printfn "Long-run proportion of up time = %f" result

    0
```

Although we defined the same simulation specs as it was in case of the activity-oriented simulation, parameter `DT` is not actually used here by Aivika. The event queue doesn't rely on the integration nodes. It has its own order of calculations concerted with the simulation in which the event queue is involved (through its `Run` method). Therefore this model is more efficient.

Here the events are created by local function `broken` and value `repaired`. The latter is just a computation that has type `Dynamics<unit>`. The former is a function that accepts one parameter. Given the start up time, this function creates a computation of type `Dynamics<unit>` too. It is called a closure. In such a way we can transfer any data we want.

```
// register a new event
do! e.Dispatch (eta (startUpTime + upTime),
        broken startUpTime)    // closure
```

The local value `started` is a computation of type `Dynamics<unit>` and it registers the very first event which must be activated at the start time of simulation. We want this computation be actuated only once. We can achieve such a behavior with help of the `once` function of module `Dynamics`.

Table 6.2: Once Function

Function and type	Description
`Dynamics.once: Dynamics<unit> -> Dynamics<unit>`	Actuates the computation only once

Since the both machines are also computations of type `Dynamics<unit>`, we can apply the `once` function to them to create a new computation that would launch the machines.

```
let e = Env ()

let machine1 = machine 1 e
let machine2 = machine 2 e

let starter =
    dynamics {
        do! machine1
        do! machine2
    } |> Dynamics.once
```

Now value `starter` is an initial point for the machines. But this computation must be involved in the main simulation through the `do!` construct that binds computations together. Please note that this binding must precede the running of the event queue.

```
let system = dynamics {

    do! starter     // start the machines
    do! e.Run       // run the event queue

    return! (totalUpTime / (2.0 * stoptime))
}
```

The rest part of the model is the same as before.

This model is more simple and clear than its activity-oriented counterpart. Also it is more precise as the up time and repair time are not rounded to be multiple of `DT`. What is also important is that the new model has a higher speed of simulation. But there is also another efficient approach. Moreover, that approach allows us to write even more concise models. I will show how we can rewrite this sample model in section 6.5, page 64.

6.4 Introducing Control Processes

In section 4.10, page 42, I told you about discrete processes. They came from mathematics. They are a generalization of function that varies in time. But in this section I would like to concern quite different processes. I will call them *control processes* to distinguish from the mathematical ones. Any control process is associated with some control flow. It is like an operating system thread or process. It is very important that the control process can be suspended at any time and then resumed later. We will use such processes to model simulation activities and I will show how the processes themselves can be easily modeled by the `DynamicsCont` workflow, which was described briefly in chapter 5.

There is an abstract class type `Process` that represents a control process. The type has the following methods and properties. The most part of them return computations. You know how to involve them in the main simulation with help of keywords `do!`, `let!` and `use!` that must be applied inside of the corresponded computation expression. Without it the computations will have no any effect.

Table 6.3: Methods and Properties on the Process type

Function and type	Description
`new:` `Env -> Process`	Creates a new control process using the specified event queue
`member Env:` `Env`	Returns the event queue which the process is associated with

`member Hold:`
`t:<float> ->`
`DynamicsCont<unit>`
	Suspends the current control process
	for the specified amount of time `t`

`member HoldD:`
`t:Dynamics<float> ->`
`DynamicsCont<unit>`
	Suspends the current control process
	for the specified amount of time `t`

`abstract Activate:` `DynamicsCont<unit>`	Activates this control process

`member IsPassivated: DynamicsCont<bool>`	Checks whether this control process is passivated
`member Passivate: DynamicsCont<unit>`	Puts the current control process to sleep until it will be awakened by some other process
`member Reactivate: DynamicsCont<unit>`	Awakens this control process
`member Start: t:float -> Dynamics<unit>`	Starts this control process at time `t`
`member StartD: t:Dynamics<float> -> Dynamics<unit>`	Starts this control process at time `t`

Please note that in the description I distinguish the *current* and *this* control processes. You can use properties `IsPassivated` and `Reactivate` from any computation in the `DynamicsCont` workflow. They don't affect the current computation and its control flow in any way. But methods `Hold`, `HoldD` and property `Passivate` that are related to the current control process should be called inside of an object value of the `Process` class type. It means that these two methods and the `Passivate` property affect the current computation in the `DynamicsCont` workflow, either suspending it or putting to sleep. Moreover, the `Passivate` property works wrong if you use it with a `Process` instance value that the control flow doesn't belong to. The process can passivate only itself.

You should never use the result of the `Activate` property directly. The corresponded computation is called automatically by methods `Start` and `StartD` that launch the control process. Certainly, the resulting computation in the `Dynamics` workflow should be involved in the main simulation. These two methods already contain a call to the `Dynamics.once` function so that the start occurs only once. The difference between the methods is in the argument type. The both methods return a more lower computation of type `Dynamics<unit>`, which allows us to easily integrate the control processes with the rest part of Aivika.

The constructor of the `Process` class type accepts the event queue. The reason is that the control processes are implemented on top of the queue. Also you can create as many control processes and then start them during the simulation as you want. The event queue is that point which binds the control processes with the main simulation after the processes are started. This scheme is very simple and it must work.

The next section illustrates how we can apply the control processes to simulate our sample model and dramatically reduce the size of the code.

6.5 Applying Process-oriented Paradigm

Under the process-oriented paradigm, we model simulation activities with help of control processes. We can easily treat such things as suspension and resumption of the control flow, request and release of the resources etc.

In Aivika we use the `DynamicsCont` workflow to model the processes themselves. I'll illustrate the idea on the same sample model which was described earlier in section 6.1, page 53, and then solved repeatedly in section 6.3, page 58. Now the solution uses the `Process` class type.

```
open System

open Maritegra.Aivika
open Maritegra.Aivika.SD

let specs = {

    StartTime=0.0; StopTime=10000.0; DT=1.0;
    Method=RungeKutta4; Randomness=SimpleRnd
}

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

let mutable totalUpTime = 0.0   // total up time for all machines

let expovariate =
    let rnd = new Random ()
    in fun lambda ->
        - log (rnd.NextDouble ()) / lambda

type Machine (e, id) =

    inherit Process (e)

    override x.Activate = dynamicscont {
        while true do
            let! startUpTime = DynamicsCont.lift time
            let upTime = expovariate upRate
            do! x.Hold (upTime)
            let! finishUpTime = DynamicsCont.lift time
            totalUpTime <- totalUpTime +
                (finishUpTime - startUpTime)
            let repairTime = expovariate repairRate
            do! x.Hold (repairTime)
    }
```

```
let env = Env ()

let machines = [| for i = 1 to 2 do yield Machine (env, i) |]

let starter =
    dynamics {
        for m in machines do
            do! m.StartD (starttime)    // launch the machine
    } |> Dynamics.once

let system = dynamics {
    do! starter     // launch the machines
    do! env.Run     // launch the environment
    return! (totalUpTime / (2.0 * stoptime))
}

[<EntryPoint>]
let main args =

    let result = Dynamics.runLast system specs
    printfn "Long-run proportion of up time = %f" result

    0
```

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

Here we create a new derived class type `Machine` and define its property `Activate`. The property returns a computation in the `DynamicsCont` workflow. This computation will be involved in the main simulation after the `starter` value is involved in computation value `system`. To optimize, we apply the `Dynamics.once` function in calculation of value `starter` that launches two machines although this optimization is not obligatory, for the `StartD` method uses the same `once` function. Also we have to launch the event queue after the machines.

What is new is that how the `Activate` property is constructed. This is an infinite loop (terminated automatically after the simulation is complete, i.e. when `time` is greater than `stoptime`) inside of which 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<float>`. As you know from section 5.2, page 50, such a computation must be *lifted* to be involved in the top computation which has another generic type `DynamicsCont`. Therefore we apply the `lift` function of module `DynamicsCont`. In such a way we get to know of the current simulation time inside of the top computation.

```
let! startUpTime = DynamicsCont.lift time
```

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

After we received the current simulation time and calculated the up time, we suspend the current control flow of the top computation.

```
do! x.Hold (upTime)
```

Please note that we have to apply the special `do!` construct to involve this action in our computation. Without it the suspension wouldn't happen. Another subtle thing is related to understanding of that the `Hold` method doesn't work with an instance value of the `Process` class type. It suspends the current *computation*, although this method is supposed to be called inside of the `Process` instance value. Each process is started with a new control flow that initially belongs to it. It is a good practice to keep the control flow inside of the process instance. Please be careful!

To simulate the both machines, we create one event queue. In general, we can always use one event queue whatever simulation would be. But you can use as many event queues as you need. Only you should involve all the computations returned by the `Run` method in the main simulation.

```
let system = dynamics {
    do! starter     // launch the machines
    do! env.Run     // launch the environment
    return! (totalUpTime / (2.0 * stoptime))
}
```

The main advantage of the process-oriented paradigm is its simplicity. All the hard work is performed by the F# compiler which generates a lot of closures. Namely through the closures, the compiler binds different parts of the code. But, fortunately, all this happens transparently for us.

The next section is devoted to that how we can manage the limited resources in our simulation model.

6.6 Managing Resources

The *resources* are usually limited. If some control process tries to acquire the depleted resource then the process is suspended until the resource is available again. After the resource is acquired and used, the control process must release it to make available for other processes.

To model such a behavior, Aivika contains the `Resource` class type. It contains a constructor and four properties, three of them return computations in the `DynamicsCont` workflow. To have effect, these computations must be involved in the simulation with help of keywords `do!` and `let!`.

Table 6.4: Methods and Properties on the Resource type

Function and type	Description
`new: e:Env * n:int -> Resource`	Creates a new resource that can be acquired `n` times without blocking
`member Env: Env`	Returns the event queue which the resource is associated with
`member N: DynamicsCont<int>`	Returns how many times the resource can be yet acquired without blocking
`member Request: DynamicsCont<unit>`	Requests the resource, which is acquired if available. Otherwise the current control process is suspended/blocked until the resource becomes available
`member Release: DynamicsCont<unit>`	Releases the resource making it available for other control processes

To suspend and resume control processes, the resource needs an event queue. Therefore the constructor requires it. The integer parameter `n` specifies how many times the resource can be immediately acquired without suspending the control process(es).

Properties `N`, `Request` and `Release` return actions that should be yet involved in the simulation. Without involving, these actions have no any effect. The `N` property requires the `let!` construct. Two others need `do!`.

The `Resource` class type is constructed in such a way that the resource can be used simultaneously by different parallel simulations. It tracks the state for each simulation separately. In general, such a capability to work simultaneously is inherent in all built-in functions and class types of Aivika, about which you can find more information in section 8.1, page 95. However, the sample models considered in this chapter mostly cannot be simulated in parallel. Section 6.10, page 80, shows how you can prepare your models for parallel simulations.

I'll illustrate the work with the resources using a sample model described in [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.

The model is as follows.

```
open System

open Maritegra.Aivika
open Maritegra.Aivika.SD

let specs = {

    StartTime=0.0; StopTime=10000.0; DT=1.0;
    Method=RungeKutta4; Randomness=SimpleRnd
}

let upRate = 1.0 / 1.0          // breakdown rate
let repairRate = 1.0 / 0.5      // repair rate

// number of times the machines have broken down
let mutable nRep = 0

// number of breakdonws in which the machine
// started repair service right away
let mutable nImmedRep = 0

// total up time for all machines
let mutable totalUpTime = 0.0

let expovariate =
    let rnd = new Random ()
    in fun lambda ->
        - log (rnd.NextDouble ()) / lambda

type Machine (e, repairPerson: Resource, id) =

    inherit Process (e)

    override x.Activate = dynamicscont {
        while true do
            let! startUpTime = DynamicsCont.lift time
            do! x.Hold (expovariate upRate)
            let! finishUpTime = DynamicsCont.lift time
            totalUpTime <- totalUpTime
```

```
                + (finishUpTime - startUpTime)
            nRep <- nRep + 1
            //
            let! n = repairPerson.N
            if n = 1 then
                nImmedRep <- nImmedRep + 1
            do! repairPerson.Request
            do! x.Hold (expovariate repairRate)
            do! repairPerson.Release
    }

let env = Env ()

let repairPerson = Resource (env, 1)

let machines =
    [| for i = 1 to 2 do
        yield Machine (env, repairPerson, i) |]

let starter =
    dynamics {
        for m in machines do
            do! m.StartD (starttime)    // launch the machine
    } |> Dynamics.once

let system = dynamics {

    do! starter         // launch the machines
    do! env.Run         // launch the environment

    let m1 = (totalUpTime / (2.0 * stoptime))
    let m2 = (float nImmedRep) / (float nRep)

    return! Dynamics.zip m1 (eta m2)
}

[<EntryPoint>]
let main args =

    let result = Dynamics.runLast system specs

    printfn "Long-run proportion of up time = %f" (fst result)
    printfn "Long-run proportion of the time when \
        immediate access to the repairperson = %f" (snd result)

    0
```

There is only one repairperson. The corresponded resource has the following declaration:

```
let repairPerson = Resource (env, 1)
```

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

```
let! n = repairPerson.N
if n = 1 then
    nImmedRep <- nImmedRep + 1
```

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

```
do! repairPerson.Request
```

After the resource is acquired, the repairing process is modeled as a short-time suspension with help of the `Hold` method.

```
do! x.Hold (expovariate repairRate)
```

After the machine is repaired, we must release the resource, i.e. free the repairperson.

```
do! repairPerson.Release
```

The rest part of the code is mostly the same as before. However, there is one subtle thing related to calculation of the second output proportion. A computation in the `Dynamics` workflow is actually a function which is repeatedly calculated in the time points. Therefore the inner value `m2` will be recalculated for each time point, which allows us to use counters `nImmedRep` and `nRep`. By the same reason we can use counter `totalUpTime` in all similar models. Only remember that using such simple counters forbids to simulate the model in parallel.

```
let system = dynamics {
    ...
    let m1 = (totalUpTime / (2.0 * stoptime))
    let m2 = (float nImmedRep) / (float nRep)

    return! Dynamics.zip m1 (eta m2)
}
```

The next section describes how we can use the resources to passivate and reactivate the control processes.

6.7 Passivating and Reactivating Processes

Each process can passivate *itself* at any time. It will be in this state until some *other* process reactivates it. To passivate, we can apply the `Passivate` property computation. The `Reactivate` property computation takes another process out of the hibernation.

The next model described in [1] illustrates the approach.

> 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.

Now we have to passivate one broken machine until the both machines are broken.

```
#nowarn "40"

open System

open Maritegra.Aivika
open Maritegra.Aivika.SD

let specs = {

    StartTime=0.0; StopTime=10000.0; DT=1.0;
    Method=RungeKutta4; Randomness=SimpleRnd
}

let upRate = 1.0 / 1.0          // breakdown rate
let repairRate = 1.0 / 0.5      // repair rate

// number of machines currently up
let mutable nUp = 0

// total up time for all machines
let mutable totalUpTime = 0.0

let expovariate =
    let rnd = new Random ()
    in fun lambda ->
        - log (rnd.NextDouble ()) / lambda

type Machine (e, repairPerson: Resource, m2: Lazy<Machine>, id) =
```

```
    inherit Process (e)

    do nUp <- nUp + 1

    override x.Activate = dynamicscont {
        while true do
            let! startUpTime = DynamicsCont.lift time
            do! x.Hold (expovariate upRate)
            let! finishUpTime = DynamicsCont.lift time
            totalUpTime <- totalUpTime
                + (finishUpTime - startUpTime)
            //
            nUp <- nUp - 1
            if nUp = 1 then
                do! x.Passivate
            else
                let! n = repairPerson.N
                if n = 1 then
                    do! m2.Value.Reactivate
            do! repairPerson.Request
            do! x.Hold (expovariate repairRate)
            nUp <- nUp + 1
            do! repairPerson.Release
    }

let env = Env ()

let repairPerson = Resource (env, 1)

let rec machines = [|
    for i = 1 to 2 do
        yield Machine (env, repairPerson,
            lazy (machines.[1 - (i - 1)]), i) |]

let starter =
    dynamics {
        for m in machines do
            do! m.StartD (starttime)    // launch the machine
    } |> Dynamics.once

let system = dynamics {

    do! starter         // launch the machines
    do! env.Run         // launch the environment

    return! (totalUpTime / (2.0 * stoptime))
```

```
}

[<EntryPoint>]
let main args =

    let result = Dynamics.runLast system specs
    printfn "The proportion of up time = %f" result

    0
```

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

```
nUp <- nUp - 1
if nUp = 1 then
    do! x.Passivate
else
    let! n = repairPerson.N
    if n = 1 then
        do! m2.Value.Reactivate
```

Also please note how we use a laziness to create the machines so that they would know of each other.

Until now we used simple numeric counters to transfer data from the DES submodel to the main computation in the `Dynamics` workflow. Nevertheless, there is a more safe approach that allows us better to integrate DES into the hybrid model.

6.8 Introducing Variables

Almost everything is represented in Aivika as computations. Even the variables of System Dynamics are computations in the `Dynamics` workflow. But sometimes we need a true variable, something that we can change imperatively. To represent such variables, Aivika contains the generic `Var` class type that has the following methods and properties.

Table 6.5: Methods and Properties on the Var<'a> type

Function and type	Description

`new:`
`e:Env * i:'a ->`
`Var<'a>`
	Creates a new variable with initial value `i`

`new:`
`e:Env * i:Dynamics<'a> ->`
`Var<'a>`
	Creates a new variable with the initial value equaled to the initial value of computation `i`

`member Env:` `Env`	Returns the event queue which the variable is associated with
`member Value:` `Dynamics<'a>`	Returns the variable value
`member SetValue:` `'a -> Dynamics<unit>`	Sets the variable value

The object value of type `Var<'a>` tracks the history of all changes. It is necessary, for the simulation time is a rather relative thing in Aivika. There can be computation branches where the time flows differently during the main simulation. To coordinate these branches, we have to use a common event queue. When some computation requests the variable for its value, our variable asks the event queue to actuate all pending events which activation time doesn't exceed the current simulation time. It allows the variable to have always the actual state under the condition that all events and all control processes that may change the variable use the same event queue. This event queue must be the first argument of the `Var` type constructor. For example, you will always have a synchronized model if you will use only one event queue for the whole simulation.

The `Value` and `SetValue` members of the class type return computations that must be yet involved in the simulation. The `Value` property computation requires the `let!` construct that allows us to know the current value of the variable. The `SetValue` method computation requires the `do!` construct to update the variable value. It is important that these members can be used in parallel simulations.

To simplify incrementing and decrementing the variable value, Aivika provides also the `Var` module with the following functions.

Table 6.6: Var module functions

Function and type	Description
`Var.env:` `Var<'a> -> Env`	Returns the event queue which the variable is associated with
`Var.value:` `Var<'a> -> Dynamics<'a>`	Returns the variable value
`Var.setValue:`	Sets the variable value

```
 'a -> Var<'a> ->
 Dynamics<unit>

 inline Var.incValue:                    Increments the variable value
 'a -> Var<'a> ->
 Dynamics<unit>
 when 'a:
 (static member (+):  'a * 'a -> 'a)

 inline Var.decValue:                    Decrements the variable value
 'a -> Var<'a> ->
 Dynamics<unit>
 when 'a:
 (static member (-):  'a * 'a -> 'a)
```

The inline functions `incValue` and `decValue` are just a syntactic sugar. The first of them wants the type to have the (+) operator. The `decValue` function requires the type with the (-) operator defined. All built-in numeric types satisfy these requirements.

In the next section I'll provide a modification of the sample model from section 6.5, page 64, rewritten with help of the generic `Var` type. Now I will confine myself to show a simple but safe generator of Fibonacci numbers, which I mentioned earlier in subsection 4.13.2, page 46. This generator is ready for parallel simulations. It returns a sequence of numbers in each time point with interval `DT`.

```
open System

open Maritegra.Aivika
open Maritegra.Aivika.SD

let specs = {

    StartTime=0.0; StopTime=1000.0; DT=1.0;
    Method=RungeKutta4; Randomness=SimpleRnd
}

let env = Env ()

let fib = Var<bigint> (env, bigint 0)

let proc =
    { new Process (env) with

        override x.Activate = dynamicscont {
```

```
            let next a = dynamicscont {

                do! DynamicsCont.lift <| fib.SetValue (a)
                do! x.HoldD (dt)
            }

            let rec loop a b = dynamicscont {

                let c = a + b

                do! next c
                do! loop b c
            }

            do! next 0I
            do! next 1I
            do! loop 0I 1I
        }
    }

let starter = proc.StartD (starttime)   // launch the process

let system = dynamics {

    do! starter         // launch the process
    do! env.Run         // launch the environment

    return! fib.Value
}

[<EntryPoint>]
let main args =

    let results = Dynamics.run system specs

    Console.WriteLine ("Fibonacci Number")
    for r in results do
        Console.WriteLine (r)

    0
```

The program returns a series of numbers in the CSV format. Perhaps the most difficult place in this model is that one which generates data:

```
let next a = dynamicscont {
```

```
    do! DynamicsCont.lift <| fib.SetValue (a)
    do! x.HoldD (dt)
}
```

Here the `next` function has type `bignint -> DynamicsCont<unit>`. The first construction generates a new Fibonacci number. The `SetValue` method returns a computation of type `Dynamics<unit>`. This computation must be lifted to be involved in the top computation of type `DynamicsCont<unit>`. After we generated a new value, we suspend the current control process for the time interval specified by the integration `DT` parameter of the simulation specs.

The `fib` variable keeps all the history of changes, where each change of the variable value corresponds to some time. Then the `Value` property returns a computation that represents all these values distributed in time and we can use it in our simulation. This is namely that mechanism with help of which the DES submodel can transfer its results to other parts of the hybrid model. To transfer data in opposite direction, we apply the `lift` function in case of the process-oriented submodel. In case of the activity-oriented and event-oriented submodels this link is even more simple.

```
let system = dynamics {

    do! starter         // launch the process
    do! env.Run         // launch the environment

    return! fib.Value
}
```

Here we ultimately return a computation representing a series of Fibonacci numbers, that is `fib.Value`. Each time some external computation requests for the number, the `fib` variable checks the event queue which this variable is associated with, i.e. `env`. The `env` queue is also associated with the `proc` control process. So, when all pending events of the queue are raised, they actuate process `proc`. This process finally updates the `fib` variable value. Thus, this variable is always in an actual state. Here we could depart slightly from our scheme and even omit the third line where the event queue is explicitly launched, for the variable does the same.

We didn't use numeric counters in this simple example. As a result, our simulation is slower than it could be, although it can be launched in parallel now. Perhaps what is more important is that data generated by the DES submodel can be used by other parts of the hybrid model. The next section is devoted to these subjects.

6.9 Using Variables and Integrating DES

The variables of generic type `Var` serve two purposes. First, they are a bridge with help of which other parts of the hybrid model can receive data from the

DES submodels. Second, the variables allow us to create models ready for parallel simulations. An idea is to replace a counter or state variable with an object value of generic type `Var` if necessary.

The next model is a rewritten version of the model described in section 6.5, page 64. It doesn't use counters any more. It is ready for parallel simulations. Its results can be used by other submodels.

```
open System

open Maritegra.Aivika
open Maritegra.Aivika.SD

let specs = {

    StartTime=0.0; StopTime=10000.0; DT=1.0;
    Method=RungeKutta4; Randomness=SimpleRnd
}

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

let expovariate =
    let rnd = new Random ()
    in fun lambda ->
        - log (rnd.NextDouble ()) / lambda

let env = Env ()

// total up time for all machines
let totalUpTime = Var<float> (env, 0.0)

type Machine (e, id) =

    inherit Process (e)

    override x.Activate = dynamicscont {
        while true do
            let! startUpTime = DynamicsCont.lift time
            let upTime = expovariate upRate
            do! x.Hold (upTime)
            let! finishUpTime = DynamicsCont.lift time
            do! DynamicsCont.lift (totalUpTime |>
                    Var.incValue (finishUpTime - startUpTime))
            let repairTime = expovariate repairRate
            do! x.Hold (repairTime)
    }
```

```
let machines =
    [| for i = 1 to 2 do
        yield Machine (env, i) |]

let starter =
    dynamics {
        for m in machines do
            do! m.StartD (starttime)    // launch the machine
    } |> Dynamics.once

let system = dynamics {
    do! starter     // launch the machines
    do! env.Run     // launch the environment
    return! (totalUpTime.Value / (2.0 * stoptime))
}

[<EntryPoint>]
let main args =

    let result = Dynamics.runLast system specs
    printfn "Long-run proportion of up time = %f" result

    0
```

Please note how we replaced the counter with a new variable with initial value 0.0. This variable means a total up time for all machines as before.

```
let totalUpTime = Var<float> (env, 0.0)
```

After the machine is broken, we have to update the variable. Now we apply the `incValue` function from module `Var`. The result must be yet lifted to be included in the top computation.

```
do! DynamicsCont.lift (totalUpTime |>
        Var.incValue (finishUpTime - startUpTime))
```

In the final computation we use the variable to find the desired proportion. Now values `totalUpTime.Value` and `stoptime` are of the same rank. The both are computations of type `Dynamics<float>`.

```
let system = dynamics {
    ...
    return! (totalUpTime.Value / (2.0 * stoptime))
}
```

If we pursue the goal to integrate DES into the hybrid model then we can use the generic `Var` type only for those variables that return the final result as in the

model above. The `totalUpTime` object value is such a variable. But it would be an overkill if you replaced all your internal counters and state variables (inside of the DES submodel) in that case if you really don't care of parallel simulations and you know exactly the order of calculations in which these variables are used. It would be reasonable to use simple variables without `Var`. The `totalUpTime` value is not that case. The order in which this variable will be used by other parts of the hybrid model is *undefined* in general case. Moreover, all previous models described in this chapter weren't ready for integration into the hybrid model. They all *should* use the `Var` generic type to return their results. But then it would be difficult for me to describe the models in simple way.

The `Var` type is not actually restricted to the purpose of integrating DES. It can be applied for integrating the agent-based submodels as well. The agents use the event queue too. They all are integrated well and form a unified scheme of modeling, where the `Dynamics` workflow is a glue that joins different parts.

6.10 Preparing Model for Parallel Simulations

Using the `Var` object values instead of different variables such as counters is fine if you want to integrate DES into the hybrid model. But it would be an overkill for internal variables of the DES submodel. However, if you care of parallel simulations then you cannot use simple variables inside of the DES submodel to save the internal state for the *same* computation during different simulations. If you can create different computations for different simulations then there is no such problem.

Fortunately, Aivika provides the generic `MemorylessVar` class type that solves the task with common computation. This type is a lightweight version of the `Var` type, which has no memory about variable's value changes but which knows about parallel execution. This version is more efficient. The generic `MemorylessVar` class type has absolutely the same interface.

Table 6.7: Methods and Properties on the MemorylessVar<'a> type

Function and type	Description

`new:`
`e:Env * i:'a ->`
`MemorylessVar<'a>`
	Creates a new variable with initial value `i`

`new:`
`e:Env * i:Dynamics<'a> ->`
`MemorylessVar<'a>`
	Creates a new variable with the initial value equaled to the initial value of computation `i`

`member Env:` `Env`	Returns the event queue which the variable is associated with

`member Value: Dynamics<'a>`	Returns the variable value
`member SetValue: 'a -> Dynamics<unit>`	Sets the variable value

Similarly, the `MemorylessVar` module has the same functions as the `Var` module.

Table 6.8: MemorylessVar module functions

Function and type	Description
`MemorylessVar.env: MemorylessVar<'a> -> Env`	Returns the event queue which the variable is associated with
`MemorylessVar.value: MemorylessVar<'a> -> Dynamics<'a>`	Returns the variable value
`MemorylessVar.setValue: 'a -> MemorylessVar<'a> -> Dynamics<unit>`	Sets the variable value
`inline MemorylessVar.incValue: 'a -> MemorylessVar<'a> -> Dynamics<unit> when 'a: (static member (+): 'a * 'a -> 'a)`	Increments the variable value
`inline MemorylessVar.decValue: 'a -> MemorylessVar<'a> -> Dynamics<unit> when 'a: (static member (-): 'a * 'a -> 'a)`	Decrements the variable value

To prepare the model for parallel simulations, just replace your internal state variables with object values of generic type `MemorylessVar`. These objects don't consume so much memory as the `Var` object values. They don't keep the history of all changes. Nevertheless, they check the current simulation time and won't allow neither requesting or updating the past values.

Certainly, another obvious approach is to create all your computations anew for each parallel simulation. Then we wouldn't need the `MemorylessVar` type at all. That approach is preferable. Unfortunately, it is not always possible.

Chapter 7

Agent-based Modeling

The agent-based modeling is quite different in comparison with DES and System Dynamics. The main entity is an *agent* which is defined as a state machine. The agents can pass messages to each other. Also we can assign the timer and timeout handlers to each active state in which the agent can be. 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.

The good news is that Aivika supports the agent-based modeling and allows you to create and use agents in your hybrid models. It is possible thanks to the fact that the agents are based on the `Dynamics` workflow as everything else in Aivika. Moreover, using such an excellent feature of F# as *object expressions* makes the definition of agents an appealing task. Like the process-oriented models of DES the agents are a result of combination of the object-oriented and functional programming.

7.1 Agents Are State Machines

The agents are implemented as state machines. The states can be nested. There are two special states: the *start* and *final* ones. All other states in which the agent can be are descendants of the start state. After the agent passes to the final state, it stops.

We can assign to each state a set of *timer* and *timeout* handlers which are active while the state itself is active. All nested states are active simultaneously. If the agent passes to another state then all handlers for the inactivated states are disabled. These handlers are such computations in the `Dynamics` workflow which are actuated in the specified amount of time. The timeout handler is actuated only once. The timer is actuated repeatedly with the specified period. This is all the difference between the timer and timeout handlers.

To functionate, each agent uses the event queue. Namely through the queue, the handlers are actuated. Also the agents can be integrated into the hybrid

model with help of the queue and the `Dynamics` workflow. In case of need the agents can use the `Var` type to transfer data to other parts of the model. It is possible thanks to the fact that almost all operations with agents are defined in the `Dynamics` workflow, which makes the integration seamless.

In general we must define the internal states for the agent and override two properties `Activate` and `Deactivate`. The both properties return computations in the `Dynamics` workflow. They have a default implementation. In the `Activate` property we can define the initial state to which the agent passes from the start state. But before it we can assign a set of timer and timeout handlers that will be active until the agent goes to the final state, i.e. while the agent is alive. The computation returned by the `Deactivate` property is actuated only before the agent passes to the final state to stop.

The abstract `Agent` class type represents the agent and has methods and properties defined in the following table, where the abstract `State` class type represents an internal state in which the agent can be.

Table 7.1: Methods and Properties on the Agent type

Function and type	Description
`new:` `e:Env -> Agent`	Creates a new agent
`member Env:` `Env`	Returns the event queue which the agent is associated with
`member StartState:` `State`	Returns the start state
`member FinalState:` `State`	Returns the final state
`member State:` `Dynamics<State>`	Returns the current innermost active state

`member SetState:`
`st:State ->`
`Dynamics<unit>`
	Sets the current innermost active state (e.g. from the timer or timeout handler)

`member SetInitialState:`
`st:State ->`
`Dynamics<unit>`
	Sets the current innermost active state (during the agent or state activation)

`abstract Activate:` `Dynamics<unit>`	Activates the agent (there is a default implementation that does nothing)

`abstract Deactivate: Dynamics<unit>`	Inactivates the agent before it goes to the final state (there is a default implementation that does nothing)
`member SetTimeout: t:float * action:Dynamics<unit> -> Dynamics<unit>`	Adds a timeout handler to the start state with computation `action` which will be actuated in time interval `t` unless the agent passes to the final state
`member SetTimeoutD: t:Dynamics<float> * action:Dynamics<unit> -> Dynamics<unit>`	Adds a timeout handler to the start state with computation `action` which will be actuated in time interval `t` unless the agent passes to the final state
`member SetTimer: t:float * action:Dynamics<unit> -> Dynamics<unit>`	Adds a timer handler to the start state with computation `action` which will be actuated in time interval `t` and then repeated with this period over and over until the agent passes to the final state
`member SetTimerD: t:Dynamics<float> * action:Dynamics<unit> -> Dynamics<unit>`	Adds a timer handler to the start state with computation `action` which will be actuated in time interval `t` and then repeated with this period over and over until the agent passes to the final state
`member Start: t:float -> Dynamics<unit>`	Starts the agent at time `t` (only once)
`member StartD: t:Dynamics<float> -> Dynamics<unit>`	Starts the agent at time `t` (only once)

You should never call the `Activate` and `Deactivate` members directly. The simulation engine will do it for you. The `Activate` member is called after the agent is started. The computation returned by the `Deactivate` member is used,

i.e. involved in the main simulation, before the agent passes to the final state.

Members `SetTimeout`, `SetTimeoutD`, `SetTimer` and `SetTimerD` allow you to assign the timeout and timer handlers to the start state of the agent. They are actuated unless the agent stops. There are the same methods for the `State` type. They all endow the agent with the internal moving force.

Members `State`, `SetState` and `SetInitialState` allow you to manage the innermost active state. The `SetInitialState` member can be applied during the agent or its state activation. The agent activation literally means an activation of its start state. This member allows you to define the next innermost active state. It is very useful if you have a state machine with nested states. The `SetState` member can be called in all other cases, for example, from the timer and timeout handlers. Actually, these two members do the same thing: they move the agent to go to the next state.

To stop the agent, you can just set the final state active.

```
dynamics {
    ...
    do! agent.SetState (agent.FinalState)
}
```

The most part of the members are computations in the `Dynamics` workflow. To take effect, they must be involved in the simulation with help of constructs `let!` and `do!` as in the example above.

In the next section I will show how we can use agents. To proceed to the example, I still have to provide a description of the abstract `State` class type before. This type has similar members and they have the same meaning. Here we can think about the agent as mainly an encapsulation of its start state. Generally, you can glance at the next table. Only please pay attention to two constructors. The first constructor creates a child state for the start state. The second constructor allows you to create the child state for any specified state. The former is just a shortcut for the latter, where the start state of the agent is used implicitly.

Table 7.2: Methods and Properties on the State type

Function and type	Description
`new:` `Agent -> State`	Creates a child state for the start state of the agent
`new:` `parent:State -> State`	Creates a child state for the specified `parent` state
`member Agent:` `Agent`	Returns the agent
`member Parent:`	Returns the parent state

`State option`

`abstract Activate: Dynamics<unit>`	Activates the state (there is a default implementation that does nothing)
`abstract Deactivate: Dynamics<unit>`	Inactivates the state (there is a default implementation that does nothing)
`member SetTimeout: t:float * action:Dynamics<unit> -> Dynamics<unit>`	Adds a timeout handler to this state with computation `action` which will be actuated in time interval `t` unless the state is inactivated
`member SetTimeoutD: t:Dynamics<float> * action:Dynamics<unit> -> Dynamics<unit>`	Adds a timeout handler to this state with computation `action` which will be actuated in time interval `t` unless the state is inactivated
`member SetTimer: t:float * action:Dynamics<unit> -> Dynamics<unit>`	Adds a timer handler to this state with computation `action` which will be actuated in time interval `t` and then repeated with this period over and over until the state is inactivated
`member SetTimerD: t:Dynamics<float> * action:Dynamics<unit> -> Dynamics<unit>`	Adds a timer handler to this state with computation `action` which will be actuated in time interval `t` and then repeated with this period over and over until the state is inactivated

---

As before, you should never call members `Activate` and `Deactivate` directly. Aivika does it each time you change the innermost active state with help of the `SetState` and `SetInitialState` methods.

The timer and timeout handlers have the same meaning as it was for the `Agent` type. Only they are assigned to the state instance. If the state is inactivated then all handlers are disabled immediately. It happens automatically. Moreover, if this state will be activated later then you will have to assign the

handlers again, usually in the `Activate` member. Also if you change the innermost active state then all intermediate states that the agent should pass to achieve the target state are activated sequentially from start to end.

The states are very lightweight. They are rather cheap. You can dynamically create new states as many times as you want, even after the agent itself is activated. Furthermore, you can dynamically create new agents and then start them at any time. It allows you to create both very simple and quite complicated models.

As regards to message passing, the agents are just objects. Therefore they can call methods of other agents. Only now the methods should generally return computations in the `Dynamics` workflow. This workflow is constructed in such a way that these computations cannot lead to suspension or stopping of the control flow of agents. Every agent can call another agent and even move it to pass to some state. There is no limitations, although it would be a good practice if only the agent itself switched to another state as this operation is low-level enough. But the agent can provide a high-level method that could be called by other agents and that would already change the state in its body.

In the next section I'll show how nicely the agents and their states can be defined using the object expressions.

7.2 Simulating Bass Diffusion Model

An agent-based version of the Bass Diffusion model is described in the AnyLogic tutorial[1].

> 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.

I will show how the same model can be simulated with help of Aivika. I will create new class type `Person` that will be derived from the `Agent` type. This agent will have two additional states: `potentialAdopter` and `adopter`. The former will be an initial state, i.e. the agent will pass to it immediately from the start state. I will assign a timeout handler to the `potentialAdopter` state. This handler will transform the potential adopter to an adopter. The time interval in which the handler is actuated depends on the advertising effectiveness. After the person becomes the adopter, he/she contacts periodically with other persons

[1] http://www.xjtek.com/anylogic/help/topic/com.xj.anylogic.help/html/ABT/Bass%20Diffusion%20Model.html

transforming them to adopters with the specified adoption fraction. To model this, I will assign already a timer handler to the `adopter` state.

```
#nowarn "40"

open System

open Maritegra.Aivika
open Maritegra.Aivika.SD

let n = 500     // the number of agents

let advertisingEffectiveness = 0.011
let contactRate = 100.0
let adoptionFraction = 0.015

let specs =
    { StartTime = 0.0; StopTime = 8.0; DT = 0.1;
      Method = RungeKutta4; Randomness = SimpleRnd }

let random =
    let rnd = new Random ()
    in fun a b -> rnd.Next (a, b)

let randomTrue =
    let rnd = new Random ()
    in fun p -> rnd.NextDouble () <= p

let expovariate =
    let rnd = new Random ()
    in fun lambda ->
        - log (rnd.NextDouble ()) / lambda

let e = Env ()

let potentialAdopters = Var (e, 0)
let adopters = Var (e, 0)

type Person () as agent =

    inherit Agent (e)

    static let mutable agents = ResizeArray<Person> ()

    let rec potentialAdopter =
        { new State (agent) with
```

```
        member x.Activate = dynamics {
            do! Var.incValue 1 potentialAdopters

            // create a timeout that will hold
            // while the state is active
            do! x.SetTimeout (expovariate
                        advertisingEffectiveness,
                    agent.SetState (adopter))
        }

        member x.Deactivate = dynamics {
            do! Var.decValue 1 potentialAdopters
        }
    }
and adopter =
    { new State (agent) with
        member x.Activate = dynamics {
            do! Var.incValue 1 adopters

            // create a timer that will hold
            // while the state is active
            do! x.SetTimerD (dynamics {
                                return expovariate
                                        contactRate
                            },
                            dynamics {
                               do! agent.SayBuy
                            })
        }

        member x.Deactivate = dynamics {
            do! Var.decValue 1 adopters
        }
    }

member private x.SayBuy = dynamics {
    do! agents.[random 0 (agents.Count - 1)].Buy
}

member private x.Buy = dynamics {
    let! st = x.State
    if st = potentialAdopter then
        if randomTrue adoptionFraction then
            do! x.SetState (adopter)
}
```

```
    override x.Activate = dynamics {
        agents.Add (x)
        do! x.SetInitialState (potentialAdopter)
    }

[<EntryPoint>]
let main args =

    let starter =
        dynamics {
            for i = 1 to n do
                let agent = Person ()
                do! agent.StartD (starttime)
        } |> Dynamics.once

    let model =
        dynamics {
            do! starter

            let! t  = time
            let! a1 = potentialAdopters.Value
            let! a2 = adopters.Value

            return (t, a1, a2)
        }

    let results = Dynamics.run model specs

    printfn "Time,Potential Adopters,Adopters"

    for (t, a1, a2) in results do
        printfn "%f,%i,%i" t a1 a2

    0
```

We track the number of all agents. Since the agents never pass to the final state, we don't define the `Deactivate` member for the agent, although it would be a good practice. In the `Activate` member we add each new agent to a static list that we use to send messages to random agents.

```
type Person () as agent =
    inherit Agent (e)

    static let mutable agents = ResizeArray<Person> ()
    ...
    override x.Activate = dynamics {
```

```
        agents.Add (x)
        do! x.SetInitialState (potentialAdopter)
    }
```

Also we see that the agent immediately passes to the `potentialAdopter` state. That state increases the number of potential adopters and assigns a timeout handler that will transform the potential adopter to an adopter. After it happens, the computation of the `Deactivate` member is used, where we decrease the number of potential adopters.

```
    let rec potentialAdopter =
        { new State (agent) with
            member x.Activate = dynamics {
                do! Var.incValue 1 potentialAdopters

                // create a timeout that will hold
                // while the state is active
                do! x.SetTimeout (expovariate
                            advertisingEffectiveness,
                        agent.SetState (adopter))
            }

            member x.Deactivate = dynamics {
                do! Var.decValue 1 potentialAdopters
            }
        }
```

The `adopter` state is similar but there is a subtle thing related to the timer. We define the time period as a computation of type `Dynamics<float>`. Such a computation is actually a function. It will be called anew every time the new period is requested. Therefore the periods will be different. Please pay a special attention to this fact. Otherwise, the simulation would be more rough, or instead we would have to use the timeout handler inside of which we would manually created repeatedly a new timeout to imitate our periodic timer. Each timeout handler can be actuated zero or one time only. The timer can be actuated zero, one or many times. All depends on that whether the state will be still active or not.

```
    and adopter =
        { new State (agent) with
            member x.Activate = dynamics {
                do! Var.incValue 1 adopters

                // create a timer that will hold
                // while the state is active
                do! x.SetTimerD (dynamics {
                                    return expovariate
```

```
                                        contactRate
                                },
                                dynamics {
                                   do! agent.SayBuy
                                })
        }

        member x.Deactivate = dynamics {
            do! Var.decValue 1 adopters
        }
    }
```

In the timer handler the agent sends to random agent a message to buy the product. Here we use the list of all activated agents. Actually, we could simplify the code and put the contents of the `SayBuy` member inside of the timer definition. The only reason why I created a separate member was that the code wouldn't be fully placed on the page of this book.

```
member private x.SayBuy = dynamics {
    do! agents.[random 0 (agents.Count - 1)].Buy
}

member private x.Buy = dynamics {
    let! st = x.State
    if st = potentialAdopter then
        if randomTrue adoptionFraction then
            do! x.SetState (adopter)
}
```

To send a message, we just call other agent's member `Buy` involving the corresponded computation in the simulation with help of the `do!` construct. In that member we check whether the innermost active state of that agent is `potentialAdopter`. If it is then we draw lots that depend on the adoption fraction and in case of success move the agent to pass to the `adopter` state. All the machinery with activation and deactivation of states is operated by Aivika.

To activate the first agents, we use the `once` function of the `Dynamics` module. It guarantees that such an activation will occur only once.

```
let starter =
    dynamics {
        for i = 1 to n do
            let agent = Person ()
            do! agent.StartD (starttime)
    } |> Dynamics.once
```

Finally, we return the results of simulation in the CSV format. Here we use the `Var` class type to generate data that depend on the agents, i.e. the counters

of potential adopters and adopters. The agents and variables use the same event queue that synchronizes all computations that pass through it.

7.3 Using Variables and Integrating Agents

The agents can be integrated into the hybrid model with help of the `Var` class type. The agent and any variable that depends on it should use one event queue. Then the agent can transfer data to other parts of the hybrid model through such variables. The scheme is the same as it was in case of DES. The previous section provides the corresponded example.

Chapter 8

Mastering Dynamics Workflow

Chapter 9

Mastering DynamicsCont Workflow

Chapter 10

Integrating with .NET Applications

Bibliography

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