IronAivika

Simulation Library for .NET

David E. Sorokin <davsor@mail.ru>
(Yoshkar-Ola, Mari El, Russia)

January 11, 2026

Contents

Introduction

I am the author of a few simulation libraries that are destined for Discrete Event Simulation and System Dynamics. *Aivika* [7] is one of them. I wrote it in the Haskell programming language.

This document *IronAivika: Simulation Library for .NET* represents my other work *IronAivika*, which is similar to Aivika, but only I wrote IronAivika in F#. This library is already focused on building and running simulation models on .NET. Earlier I called it Aivika too, but to make a distinguishing, I decided to rename the library for .NET and make a new name similar to IronPython, IronScheme and so on.

I use the IronAivika library in my visual simulation tool VisualAivika [8]. IronAivika can be useful for you if you will decide to write extension modules in .NET for your VisualAivika models to use such modules within simulation. Also you can use IronAivika without VisualAivika as a standalone self-sufficient simulation library.

Probably, the most exiciting thing is that VisualAivika allows exporting simulation models as .NET applications, which can be run without VisualAivika itself, where all the programming code of such exported simulation models is provided in the source form. Once the simulation model is exported, it is up to you how you will compile and run your models.

Chapter 1

Getting Started

Before we build our first simulation model, we have to introduce some basic concepts that lie in foundation of the IronAivika simulation approach. They widely use a notion of abstract computation and their practical usability essential depends on the special feature of the F# programming language, which is known as *computation expressions*.

If you are already familiar with the F# asynchronous workflow then you can find the IronAivika approach simple and easy to use. Otherwise, it is strongly recommended that you should be acquainted with the mentioned asynchronous workflow first.

1.1 Simulation

We can treat the simulation as a function of the simulation run:

```
namespace Simulation.Aivika

type Simulation<'a> = Simulation of (Run -> 'a)
```

Here the `Run` type denotes some object that contains the information about the current simulation run. Its definition is quite implementation dependent.

There is also a computation expression builder that allows us to create `Simulation` computations. The builder has name `simulation`:

```
let x : Simulation<'a> = simulation { .. }
```

Given the specified simulation `Specs`, we can create a simulation `Run` and then launch the simulation to receive the result:

```
module Simulation =

    val run : Specs -> Simulation<'a> -> 'a
    val runSeries : int -> Specs -> Simulation<'a> -> seq<Async<'a>>
```

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

```
type Time = float
type Method = Euler | RungeKutta2 | RungeKutta4

type Specs =
    { StartTime : Time;
      StopTime : Time;
      DT : Time;
      Method : Method;
      GeneratorType : GeneratorType }
```

The omitted `GeneratorType` type allows specifying the random number generator.

We can use the same seed with the `SimpleGeneratorWithSeed` data constructor to get always a reproducible sequence of numbers, which can be helpful for testing. Below we will use the `StrongGenerator` data constructor to get a random number generator of high quality. But you can also use the `SimpleGenerator` data constructor. It is quite fast on the Windows platform, but it may return random numbers of rather poor quality, although it depends on the platform. For example, the random number generator works much slower on Linux.

Returning to the main topic, the main idea is that many simulation models can be ultimately reduced to the `Simulation` computation. Hence they can be trivially simulated using the mentioned above run functions by the specified specs. At least, all models that we will build with help of IronAivika are such ones.

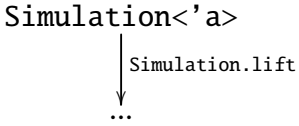
Now the following concept may look difficult at a glance, but it is very important for understanding. All simulation computations in IronAivika

are interconnected. They can be either transformed to others or can be run within others. The transformation of computation is usually called *lifting* in functional programming.

IronAivika defines a set of inline functions that all have a common name `lift`. The following function allows transforming an arbitrary `Simulation` computation to something equivalent but defined as another computation denoted below as awkward type with letter *m*.

```
module Simulation =
   val inline lift : Simulation<'a> -> ^m
```

It allows us to transform the arbitrary `Simulation` computation to anything else:



1.2 External Parameters

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

To represent such parameters, IronAivika uses almost the same definition that it uses for representing the `Simulation` computation.

```
type Parameter<'a> = Parameter of (Run -> 'a)
```

The corresponding computation builder is called `parameter`:

```
let x: Parameter<'a> = parameter { .. }
```

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

```
module Parameter =
    val memo : Parameter<'a> -> Parameter<'a>
```

We usually have to memoize the parameter if its computation is impure and it depends on performing some side effect such as reading an external file or generating a random number.

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

```
module Parameter =

    val starttime : Parameter<Time>
    val stoptime : Parameter<Time>
    val dt : Parameter<Time>
```

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

```
module Parameter =

    val randomUniform : float -> float -> Parameter<float>
    val randomNormal : float -> float -> Parameter<float>
    val randomExponential : float -> Parameter<float>
    val randomErlang : float -> int -> Parameter<float>
    val randomPoisson : float -> Parameter<int>
    val randomBinomial : float -> int -> Parameter<int>
```

To support the Design of Experiments (DoE), IronAivika defines two additional computations that return the current simulation run index and the total run count respectively, when launching the Monte-Carlo simulation.

```
module Parameter =

    val runIndex : Parameter<int>
    val runCount : Parameter<int>
```

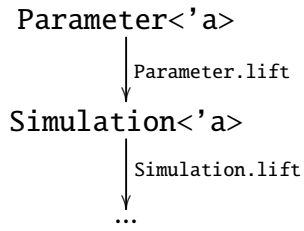
As before, there is the `lift` function that allows us to transform the arbitrary `Parameter` computation.

```
module Parameter =
   val inline lift : Parameter<'a> -> ^m
```

For example, the arbitrary `Parameter` computation can be transformed to the `Simulation` one. It means that the former can be used in every piece of the code, where the `Simulation` computation is expected. It is just enough to call the `lift` function:

```
let x1 : Parameter<'a>  = ...
let x2 : Simulation<'a> = x1 |> Parameter.lift
```

It allows using the external parameters within the simulation:



1.3 Ordinary Differential Equations

Although the main strength of the IronAivika library is its orientation on Discrete Event Simulation, it is simpler to demonstrate the method using another paradigm. So, our first simulation model will be described by a set of Ordinary Differential Equations (ODEs).

The Equation Compiler version of VisualAivika uses another approach for integrating the ODE equations, which is highly optimized for speed of execution. But the Interpreter version of VisualAivika uses the approach described here.

Assuming that the `Point` type represents a modeling time point within the current simulation run, we can define a time varying function, which would be suitable for approximating the integrals.

Let us call it the `Dynamics` computation to emphasize that it can model some dynamic processes defined usually with help of differential equations and difference equations of System Dynamics.

```
type Dynamics<'a> = Dynamics of (Point -> 'a)
```

The corresponding computation builder is called `dynamics`:

```
let x: Dynamics<'a> = dynamics { .. }
```

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

```
module Dynamics =
    val time : Dynamics<Time>
```

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

```
module Dynamics =

    val runInStartTime : Dynamics<'a> -> Simulation<'a>
    val runInStopTime : Dynamics<'a> -> Simulation<'a>

    val runInIntegTimes : Dynamics<'a> -> Simulation<seq<'a>>
    val runInTimes : seq<Time> -> Dynamics<'a> -> Simulation<seq<'a>>
```

The key feature of the `Dynamics` computation is that it allows us to approximate the integral by the specified derivative and initial value:

```
module SD =
    val integ : Lazy<Dynamics<float>>
                -> Dynamics<float>
                -> Dynamics<float>
```

The point is that the ordinary differential and difference equations can be defined declaratively almost as in maths and as in many commercial simulation software tools of System Dynamics such as Vensim[12], ithink/Stella[2], Berkeley-Madonna[3] and the mentioned VisualAivika.

IronAivika overloads arithmetic operators for type `Dynamics<float>`. It allows us to treat these computations as numbers.

Moreover, we can create new computations from real numbers:

```
module SD =
    val num : 'a -> Dynamics<'a>
```

Actually, this is a synonym of the `dynamics.Return` method, but the former is more convenient for using within differential equations as it will be shown below.

To demonstrate the approach, we can rewrite a model from the 5-Minute Tutorial of Berkeley-Madonna[3] with the following equations.

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

For example, we can return the integral values in the final simulation time. In the same way, we could return the integral values in arbitrary time points that we would specify using other run functions.

But there is another way. We can return the `ResultSet` object that encompasses all results we are interested in.

Provided that the library implementation files are referenced properly, we can write the following complete script with the definition of our simulation model in file `Model.fsx`.

```
// File ChemicalReaction/Model.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

#nowarn "40"

open System

open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.SD

let specs = {

    StartTime=0.0; StopTime=13.0; DT=0.01;
    Method=RungeKutta4; GeneratorType=StrongGenerator
}

let model : Simulation<ResultSet> = simulation {
```

```
    let rec a = integ (lazy (- ka * a)) (num 100.0)
    and b = integ (lazy (ka * a - kb * b)) (num 0.0)
    and c = integ (lazy (kb * b)) (num 0.0)
    and ka = 1.0
    and kb = 1.0

    return
        [ResultSource.From ("A", a, "Var A");
         ResultSource.From ("B", b, "Var B");
         ResultSource.From ("C", c, "Var C")]
            |> ResultSet.create
}
```

To show the results in the final time point, for example, we can call the corresponding function.

We can write the code in another file `Run.fs`, which will allow us to use the model repeatedely for another goal as we will see soon.

```
// File ChemicalReaction/Run.fsx

#load "Model.fsx"

open Model

open Simulation.Aivika
open Simulation.Aivika.Results

ResultSet.printInStopTime Model.specs Model.model
```

It prints the following information in terminal:

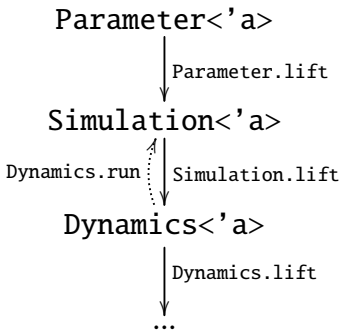
```
----------

// time
t = 13

// Var A
A = 0,0002260329409450 24

// Var B
B = 0,00293842823104866

// Var C
C = 99,9968355388281
```

Like that it was true for other computations, there is a transforming function for the `Dynamics` computation as well.

```
module Dynamics =
   val inline lift : Dynamics<'a> -> ^m
```

It allows using the arbitrary `Dynamics` computations is those places, where something different is expected. It is worth noting that the F# compiler checks the types and it will not allow transforming the computations if it makes no sense. So, the lifting functions are quite safe.

The `Parameter` and `Simulation` computations can be transformed to the `Dynamics` one:



1.4 Simulation Experiment

The model constructing is very important by no means, but it is not sufficient. To validate the model or to analyze it, we have to automate the process of displaying the most important simulation results. IronAivika provides with such an ability.

The simulation library allows saving the results in CSV files that can be then opened in the Office application or R statistics tool for the further analysis. Also the library allows plotting the results on charts as well as plotting histograms by the collected statistics.

One of the important charts is so called the Deviation Chart that plots the trend and confidence intervals by rule *3-sigma* (based on Chebyshev's inequality). There are also Time Series and XY Chart that draw the simulation results for each run, while the Deviation Chart is cumulative and it is displayed for the whole Monte-Carlo simulation experiment.

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

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

We define an `Experiment` object specifying the simulation specs and a number of runs.

```
let experiment = Experiment ()

experiment.Specs <- Model.specs
experiment.RunCount <- 1
```

Then we define the `IExperimentProvider` providers that already know how to render the results.

```
let provider1 = ExperimentSpecsProvider ()
let provider2 = TimeSeriesProvider ()
let provider3 = LastValueProvider ()
let provider4 = TableProvider ()
```

The most important property of the provider is `Series` or something different with similar name.

So, we could ask the Time Series provider to render three variables `A`, `B` and `C` that we return from the model.

```
provider2.Series <-
    [ ResultSet.findByName "A";
      ResultSet.findByName "B";
      ResultSet.findByName "C"]
        |> ResultTransform.concat
```

By default, many providers render all variables that are returned by the model. Therefore, we could omit this assignment statement.

Then we ask the experiment to render HTML by the specified model, which will use our providers.

```
experiment.RenderHtml (Model.model, providers)
    |> Async.RunSynchronously
```

Below is stated a complete simulation experiment script[1] written in F#.

```
// File ChemicalReaction/RunExperiment.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.dll"

#load "Model.fsx"

open Model

open System
open System.IO

open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Web

let experiment = Experiment ()

experiment.Specs <- Model.specs
experiment.RunCount <- 1

let provider1 = ExperimentSpecsProvider ()
let provider2 = TimeSeriesProvider ()
let provider3 = LastValueProvider ()
let provider4 = TableProvider ()

let providers =
    [ provider1 :> IExperimentProvider<TextWriter>;
      provider2 :> IExperimentProvider<TextWriter>;
      provider3 :> IExperimentProvider<TextWriter>;
      provider4 :> IExperimentProvider<TextWriter> ]

experiment.RenderHtml (Model.model, providers)
    |> Async.RunSynchronously
```

[1] On Linux and macOS we have to replace the Windows-based assembly `Simulation.Aivika.Charting.dll` with its GTK or Skia version.

When running this script, we receive in the terminal of Microsoft Windows something like this[2]

```
C:\Docs\Projects\Aivika\examples\ChemicalReaction>fsi RunExperiment.fsx
Updating directory experiment
Generated file experiment\TimeSeries(1).png
Generated file experiment\Table(1).csv
Generated file index.html
```

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

The Web page shows the simulation experiments specs, the time series chart, the last values and provides with a hyper-link to load the CSV file with the results.

[2]Here and below the examples use the outdated call to the `fsi` program. Now it should be replaced with the `dotnet fsi` call as illustrated in section 9.6.

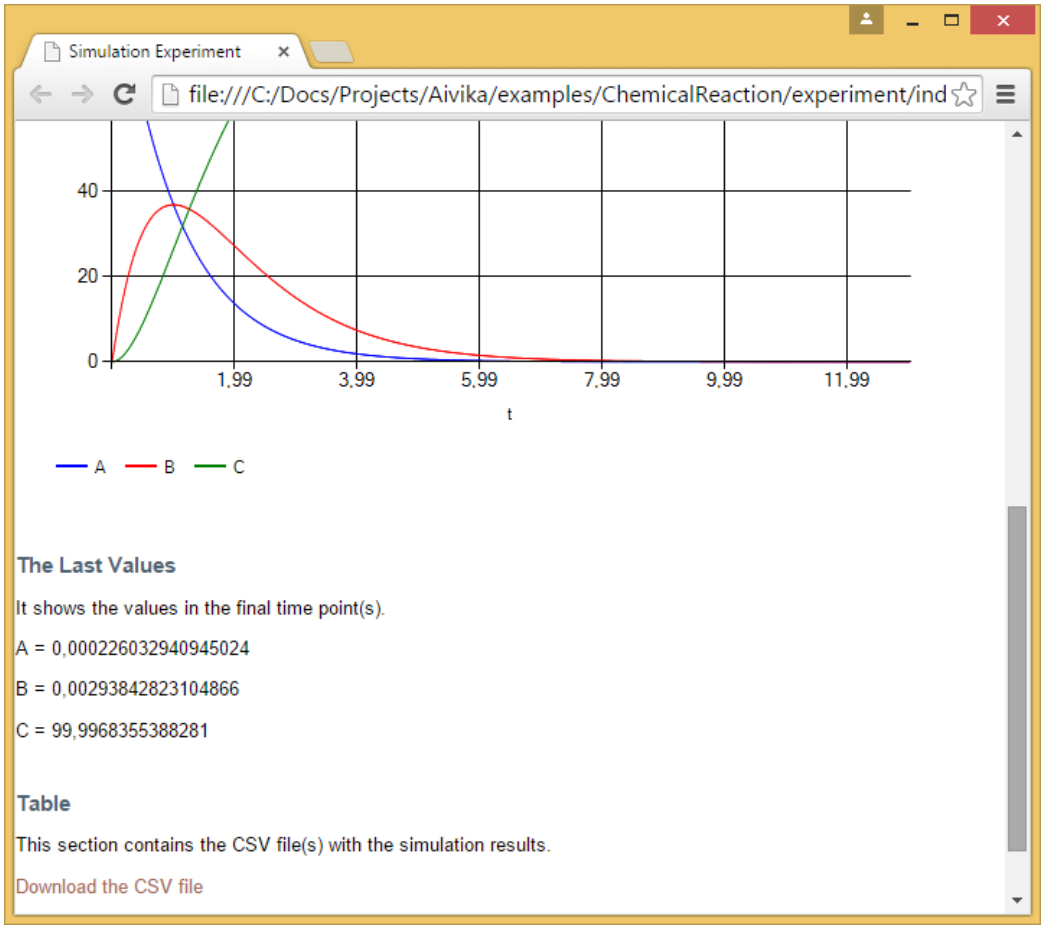


Figure 1.1: The rendered simulation experiment in the Internet browser.

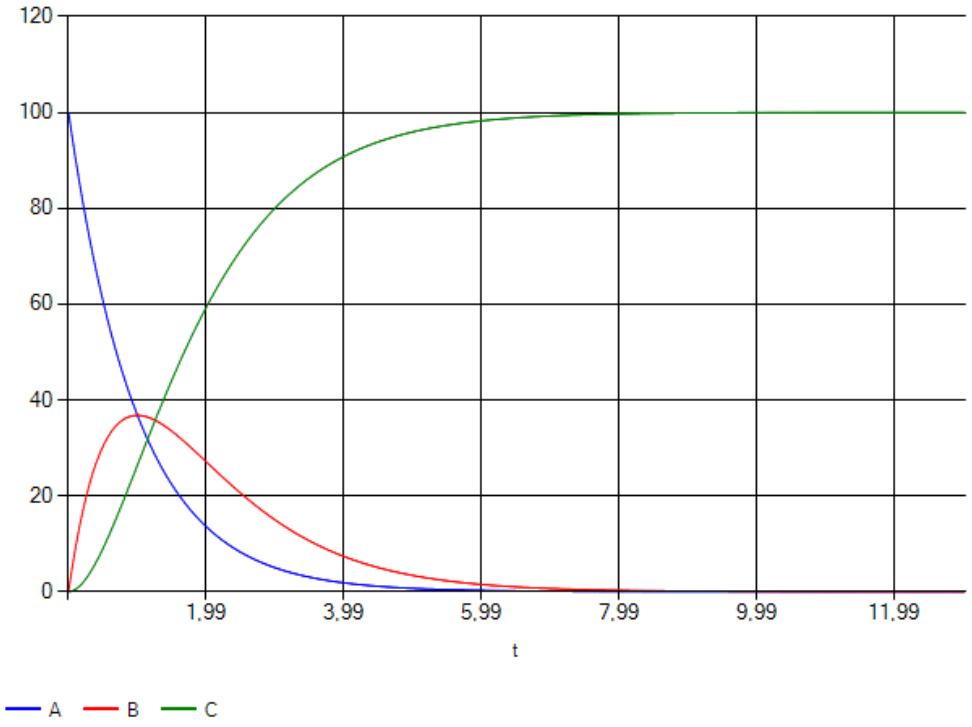


Figure 1.2: The time series chart for chemical reaction.

Chapter 2

Discrete Event Simulation

The main focus of the IronAivika library is Discrete Event Simulation (DES). Below are described basic ideas used in IronAivika for formalizing the simulation model and reasoning in terms of this paradigm.

2.1 Event-oriented Simulation

Under the *event-oriented* paradigm[5, 4] of DES, we put all pending events in the priority queue, where the first event has the minimal activation time. Then we sequentially activate the events removing them from the queue. During such an activation we can add new events. This scheme is also called *event-driven*.

Here we can use almost the same time-varying function for the event-oriented simulation, which we used for approximating the integrals with help of the `Dynamics` workflow.

```
namespace Simulation.Aivika

type Eventive<'a> = Eventive of (Point -> 'a)
```

The difference is that we can strongly guarantee[1] on level of the type system of F# that the `Eventive` computation is always synchronized with the event queue. Here we imply that every simulation run has an internal event queue, which is contained in the `Run` object.

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

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

```
module Eventive =
    val enqueue : Time -> Eventive<unit> -> Eventive<unit>
```

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

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

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

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

```
module Eventive =
    val run : Eventive<'a> -> Dynamics<'a>
```

The corresponding computation builder has name `eventive`:

```
let x : Eventive<'a> = eventive { .. }
```

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

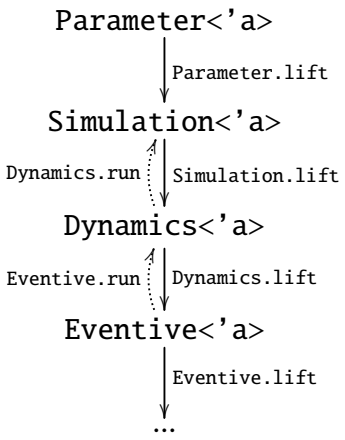
Some other run functions are destined for the most important use cases, when we can run the input computation directly within `Simulation` in the initial and final modeling time points, respectively.

```
module Eventive =

    val runInStartTime : Eventive<'a> -> Simulation<'a>
    val runInStopTime : Eventive<'a> -> Simulation<'a>
```

Following the rule, the arbitrary `Dynamics` computation can be transformed to the `Eventive` computation. The latter in its turn can be transformed to another one with help of the corresponding `lift` function.

```
module Eventive =
   val inline lift : Eventive<'a> -> ^m
```

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



2.2 Mutable Reference

Many DES models need a mutable reference. F# already provides with the `ref` reference type. We might use it in the sequential simulation model, nevertheless, it should be replaced with the following `Ref` type under simple obligations: the reference must be created within `Simulation` computation and then be used within `Eventive`, or another computation to which the latter can be transformed with help of the `lift` function.

```
type Ref<'a>
```

```
module Ref =

    val create : 'a -> Simulation<Ref<'a>>

    val read : Ref<'a> -> Eventive<'a>
    val write : 'a -> Ref<'a> -> Eventive<unit>
    val modify : ('a -> 'a) -> Ref<'a> -> Eventive<unit>

    val inc : Ref<int> -> Eventive<unit>
    val dec : Ref<int> -> Eventive<unit>
```

The standard `ref` reference of F# can be safely replaced with the `Ref` type, but in this document we will prefer the former for simplicity, although it is recommended to the use the latter instead.

In the future, other simulation modes can be added to IronAivika like that one how it is done in Aivika, where the standard `ref` reference would not work already. By using the `Ref` type, it would be then easier to migrate your code for such new simulation modes.

2.3 Example: Event-oriented Simulation

The IronAivika distribution contains examples of using the mutable references in the DES models, one of which is provided below. The task itself is described in the documentation of SimPy[4].

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

The corresponding model is as follows.

```
// File MachRep1EventDriven/Model.fsx

#nowarn "40"

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
```

```
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

open Simulation.Aivika
open Simulation.Aivika.Results

let specs = {

    StartTime=0.0; StopTime=10000.0; DT=0.05;
    Method=RungeKutta4; GeneratorType=StrongGenerator
}

let meanUpTime = 1.0
let meanRepairTime = 0.5

let model = simulation {

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

    let rec machineBroken startUpTime =
        eventive {

            // the machine is broken

            let! finishUpTime =
                Dynamics.time |> Dynamics.lift

            totalUpTime := !totalUpTime
                + (finishUpTime - startUpTime)

            let! repairTime =
                Parameter.randomExponential meanRepairTime
                    |> Parameter.lift

            // register a new event

            do! machineRepaired
                    |> Eventive.enqueue
                        (finishUpTime + repairTime)
        }
    and machineRepaired =
        eventive {

            // the machine is repaired
```

```
            let! startUpTime =
                Dynamics.time |> Dynamics.lift

            let! upTime =
                Parameter.randomExponential meanUpTime
                    |> Parameter.lift

            // register a new event

            do! machineBroken startUpTime
                    |> Eventive.enqueue
                        (startUpTime + upTime)
        }

    do! machineRepaired |> Eventive.runInStartTime
    do! machineRepaired |> Eventive.runInStartTime

    let upTimeProp =
        eventive {
            let! t = Dynamics.time |> Dynamics.lift
            return (!totalUpTime / (2.0 * t))
        }

    return
        [ResultSource.From ("upTimeProp", upTimeProp,
            "Long-run proportion of up time \
            (must be about 0.66)")]
                |> ResultSet.create
}
```

In the simplest case we can use the next simulation experiment that shows the results in the final simulation time.

```
#load "Model.fsx"

open Model

open Simulation.Aivika
open Simulation.Aivika.Results

ResultSet.printInStopTime Model.specs Model.model
```

When running it, we receive in the terminal of macOS something like this

```
bash-3.2$ fsharpi Run.fsx
----------

// time
t = 10000

// Long-run proportion of up time (must be about 0.66)
upTimeProp = 0,661181942118896
```

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

2.4 Variable with Memory

Sometimes we need an analog of the mutable reference that would save the history of its values. IronAivika defines the corresponding `Var` type. It has almost the same functions with similar type signatures, which the `Ref` reference has.

```
type Var<'a>

module Var =

    val create : 'a -> Simulation<Var<'a>>

    val read : Var<'a> -> Eventive<'a>
    val write : 'a -> Var<'a> -> Eventive<unit>
    val modify : ('a -> 'a) -> Var<'a> -> Eventive<unit>

    val inc : Var<int> -> Eventive<unit>
    val dec : Var<int> -> Eventive<unit>
```

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

```
module Var =
    val memo : Var<'a> -> Dynamics<'a>
```

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

On the contrary, the `read` function returns a computation of the *recent* actual value for the *current* simulation time point. This value is already destined to be used in the discrete event simulation as it is synchronized with the event queue by the very design of the library. Such is the `Eventive` computation, which must be synchronized with the event queue.

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

```
module Var =
    val freeze : Var<'a> -> Eventive<Time [] * 'a [] * 'a []>
```

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

As a caution, try to avoid using the `Var` variable as it is rather slow in comparison to the standard F# reference and even the `Ref` reference. The usual mistake is to use the variable for accumulating the statistics, but IronAivika contains optimized data structures designed especially for this task as described in chapter 5.

The `Var` variable is destined for combining the discrete event simulation with ordinary differential equations and difference equations. It should be used mainly in such a way.

2.5 Process-oriented Simulation

Under the *process-oriented* paradigm[5, 4], we model simulation activities with help of a special kind of processes. We can explicitly suspend and resume such processes. Also we can request for and release of the resources implicitly suspending and resuming the processes in case of need.

IronAivika actually supports the process-oriented simulation, at least, on two different levels. A higher level, which uses streams of data and blocks that operate on transacts created by these streams, is considered

further. Below is described a lower level, which is a foundation for the higher level, nevertheless.

To model the process, IronAivika uses the following type as a basis.

```
type Cont<'a> = Cont of (('a -> Eventive<unit>) -> Eventive<unit>)
```

The corresponding computation builder has name `cont`:

```
let x : Cont<'a> = cont { .. }
```

It is known from the theory of functional programming that we can suspend the `Cont` computation and then resume later. This is one of the main features that distinguishes this computation, being based on so called *continuations*.

The key idea is that the value of type `Cont<unit>` can be reduced to a function of type `Eventive<unit> -> Eventive<unit>`, which is actually the end part of the type signature for the `Eventive.enqueue` function mentioned above. That function enqueues a new event with the desired time of actuating the event handler.

It means that we can take an arbitrary computation of type `Cont<unit>`, suspend it and then resume it at another modeling time with help of the event queue.

This technique allows us to hold the process for the specified time interval. But sometimes we need to passivate the process for indefinite time so that another simulation activity could reactivate it later.

Therefore, we need some data structure to store the continuation that we would receive within the `Cont` computation. The process identifier `ProcId` can play a role of such data structure.

```
data ProcId

module Proc =
    val createId : Simulation<ProcId>
```

Then the *discontinuous process* can be represented with help of the following computation.

```
type Proc<'a> = Proc of (ProcId -> Cont<'a>)
```

The corresponding computation builder has name `proc`:

```
let x : Proc<'a> = proc { .. }
```

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

```
module Proc =

    val run : Proc<unit> -> Eventive<unit>
    val runUsingId : ProcId -> Proc<unit> -> Eventive<unit>

    val runInStartTime : Proc<unit> -> Simulation<unit>
    val runInStartTimeUsingId : ProcId -> Proc<unit> -> Simulation<unit>

    val runInStopTime : Proc<unit> -> Simulation<unit>
    val runInStopTimeUsingId : ProcId -> Proc<unit> -> Simulation<unit>
```

If the process identifier is not specified then a new generated identifier is assigned, when running the process. Every process has always its own unique identifier.

```
module Proc =
    val id : Proc<ProcId>
```

In case of need we can run a sub-process using another identifier.

```
module Proc =
    val usingId : ProcId -> Proc<'a> -> Proc<'a>
```

The characteristic feature of the `Proc` computation is that the process can be hold for the specified time interval through the event queue, following the approach described above in this section.

```
module Proc =
    val hold : Time -> Proc<unit>
```

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

```
module Proc =

    val interrupt : ProcId -> Eventive<unit>
    val isInterrupted : ProcId -> Eventive<bool>
```

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

To passivate the process for indefinite time to reactive it later, we can use the following functions.

```
module Proc =

    val passivate : Proc<unit>
    val isPassivated : ProcId -> Eventive<bool>
    val reactivate : ProcId -> Eventive<unit>
```

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

```
module Proc =

    val cancelUsingId : ProcId -> Eventive<unit>
    val cancel<'a> : Proc<'a>
    val isCancelled : ProcId -> Eventive<bool>
```

Sometimes we need to run the arbitrary sub-process within the specified time-out.

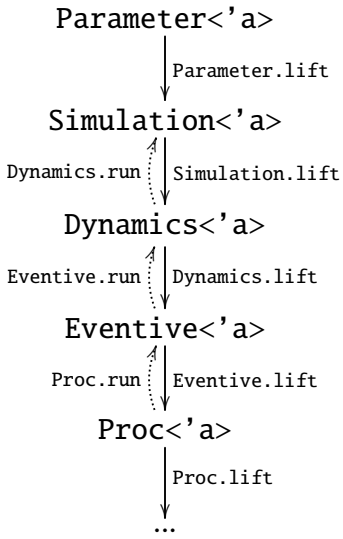
```
module Proc =
    val timeout : Time -> Proc<'a> -> Proc<'a option>
```

If the sub-process executes too long and exceeds the time limit, then it is immediately canceled and `None` is returned within the `Proc` computation. Otherwise, the computed result is returned right after it is received by the sub-process.

Every simulation computation that we considered earlier can be transformed to the `Proc` computation, which in its turn can be transformed to another one with help of the corresponding `lift` function, at least, it can be transformed to itself.

```
module Proc =
   val inline lift : Proc<'a> -> ^m
```

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



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

```
((p : Proc<unit>) |> Proc.run |> Eventive.lift) : Proc<unit>
```

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

```
module Proc =

    val spawn : Proc<unit> -> Proc<unit>
    val spawnWith : ContCancellation -> Proc<unit> -> Proc<unit>
```

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

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

The stated above `timeout` function uses `spawnWith` to run the specified sub-process within time-out.

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

```
module Proc =

    val par : Proc<'a> list -> Proc<'a list>
    val par_ : Proc<'a> list -> Proc<unit>
```

The `Proc` computation can be memoized so that the resulting process would always return the same value within the simulation run regardless of that how often the process was requested repeatedly.

```
module Proc =
    val memo : Proc<'a> -> Proc<'a>
```

Using the random number generator and the `hold` function, we can model an activity that is performed for some random time, for example, a processing of the item by the machine tool.

IronAivika contains a set of built-in random activities, which all are defined in the following way.

```
module Proc =

    let randomUniform minimum maximum = proc {

        let! t = Parameter.randomUniform minimum maximum |> Parameter.lift
        do! hold t
        return t
    }

    let randomUniform_ minimum maximum = proc {

        let! t = Parameter.randomUniform minimum maximum |> Parameter.lift
        do! hold t
    }
```

The first function holds the current discontinuous process for a random time interval distributed uniformly and then returns that interval within the computation. The second function performs a side effect only without returning the interval.

Below is provided a list of predefined random activities that hold the current process for a random time interval according to their distributions.

```
module Proc =

    val randomUniform: minimum:float -> maximum:float -> Proc<float>
    val randomUniform_: minimum:float -> maximum:float -> Proc<unit>

    val randomUniformInt: minimum:int -> maximum:int -> Proc<int>
    val randomUniformInt_: minimum:int -> maximum:int -> Proc<unit>

    val randomNormal: mean:float -> deviation:float -> Proc<float>
    val randomNormal_: mean:float -> deviation:float -> Proc<unit>

    val randomExponential: mean:float -> Proc<float>
    val randomExponential_: mean:float -> Proc<unit>

    val randomErlang: beta:float -> m:int -> Proc<float>
    val randomErlang_: beta:float -> m:int -> Proc<unit>

    val randomPoisson: mean:float -> Proc<int>
    val randomPoisson_: mean:float -> Proc<unit>

    val randomBinomial: prob:float -> trials:int -> Proc<int>
    val randomBinomial_: prob:float -> trials:int -> Proc<unit>
```

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

2.6 Example: Process-oriented Simulation

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

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

Using the processes, we can solve the task in a more elegant way.

```
// File MachRep1/Model.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

open System

open Simulation.Aivika
open Simulation.Aivika.Results

let specs = {

    StartTime=0.0; StopTime=1000.0; DT=1.0;
    Method=RungeKutta4; GeneratorType=StrongGenerator
}

let meanUpTime = 1.0
let meanRepairTime = 0.5

let model = simulation {

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

    let machine = proc {

        while true do

            let! upTime = Proc.randomExponential meanUpTime
            totalUpTime := !totalUpTime + upTime

            do! Proc.randomExponential_ meanRepairTime
    }
```

```
    do! Proc.runInStartTime machine
    do! Proc.runInStartTime machine

    let upTimeProp =
        eventive {
            let! t = Dynamics.time |> Dynamics.lift
            return (!totalUpTime / (2.0 * t))
        }

    return
        [ResultSource.From ("upTimeProp", upTimeProp,
            "Long-run proportion of up time \
            (must be about 0.66)")]
                |> ResultSet.create
}
```

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

Now we will launch the Monte-Carlo simulation with 1000 simultaneous runs. After running the experiment, we will receive the deviation chart, statistics summary and histogram for our single variable that our `model` returns within `Simulation`.

```
// File MachRep1/RunExperiment.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.dll"

#load "Model.fsx"

open Model

open System
open System.IO

open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
```

```
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Web

let experiment = Experiment ()

experiment.Specs <- Model.specs
experiment.RunCount <- 1000

let provider1 = ExperimentSpecsProvider ()
let provider2 = DeviationChartProvider ()
let provider3 = LastValueStatsProvider ()
let provider4 = LastValueHistogramProvider ()

let providers =
    [ provider1 :> IExperimentProvider<TextWriter>;
      provider2 :> IExperimentProvider<TextWriter>;
      provider3 :> IExperimentProvider<TextWriter>;
      provider4 :> IExperimentProvider<TextWriter> ]

experiment.RenderHtml (Model.model, providers)
    |> Async.RunSynchronously
```

Figure 2.1 shows the deviation chart, but figure 2.2 displays the histogram by data collected, when running 1000 simulation runs.

The statistics summary is contained in the generated `index.html` file. In our case it is stated in table 2.1. The actual numbers may differ from experiment to experiment, but the tendency will be always the same.

Table 2.1: The summary for the long-run proportion of up time.

upTimeProp	
mean	0.666144947122581
deviation	0.00884072527833155
minimum	0.63948370536354
maximum	0.692046918776428
count	1000

There is also another popular paradigm applied to Discrete Event Simulation. It usually gives more rough simulation results, for we have to scale

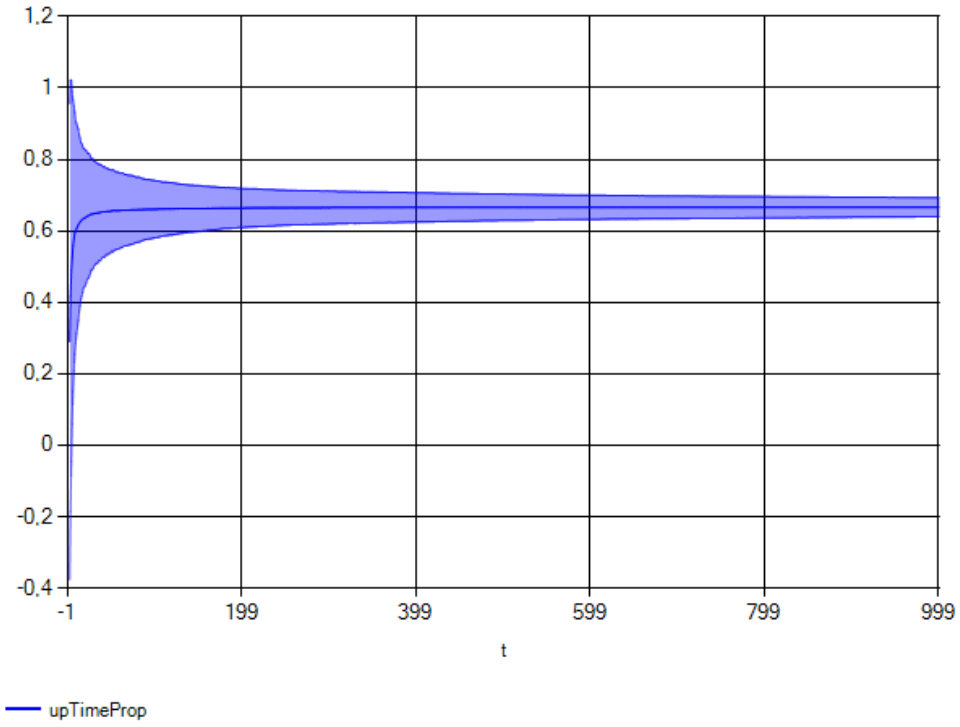


Figure 2.1: The deviation chart for the long-run proportion of up time.

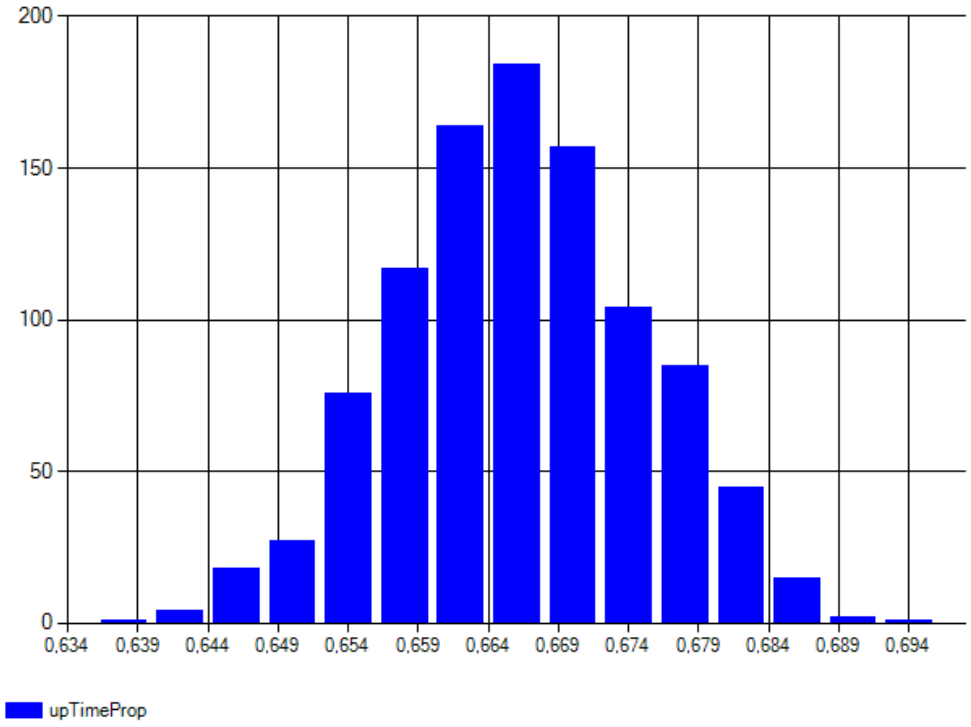


Figure 2.2: The long-run proportion histogram of up time.

the modeling time. The next two sections show how IronAivika supports that paradigm and how we can apply it to solve the same task.

2.7 Activity-oriented Simulation

Under the *activity-oriented* paradigm[5, 4] of DES, we break time into tiny increments. At each time point, we look around at all the activities and check for the possible occurrence of events. Sometimes this scheme is called *time-driven*.

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

```
module Eventive =
    val enqueueWithTimes: #seq<Time> -> Eventive<unit> -> Eventive<unit>
```

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

```
module Eventive =
    val enqueueWithIntegTimes: Eventive<unit> -> Eventive<unit>
```

Being defined in such a way, the activity-oriented simulation can be combined with the event-oriented and process-oriented ones.

2.8 Example: Activity-oriented Simulation

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

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

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

```
// File MachRep1ActivityOriented/Model.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

open System

open Simulation.Aivika
open Simulation.Aivika.Results

let specs = {

    StartTime=0.0; StopTime=1000.0; DT=0.05;
    Method=RungeKutta4; GeneratorType=StrongGenerator
}

let meanUpTime = 1.0
let meanRepairTime = 0.5

let model = simulation {

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

    let machine () =

        // a number of iterations when working
        let upNum = ref -1

        // a number of iterations when prepairing
        let repairNum = ref -1

        // the start up time
        let startUpTime = ref 0.0

        // wait for the break
        let untilBroken = eventive {
            decr upNum
        }
```

```
// wait for the repair
let untilRepaired = eventive {
    decr repairNum
}

// when the tool is broken
let broken = eventive {

    decr upNum

    // the machine is broken

    let! t = Dynamics.time |> Dynamics.lift
    let! dt = Parameter.dt |> Parameter.lift
    let! repairTime =
        Parameter.randomExponential meanRepairTime
            |> Parameter.lift

    totalUpTime := !totalUpTime
        + (t - !startUpTime)
    repairNum := int (repairTime / dt)
}

// when the tool is repaired
let repaired = eventive {

    decr repairNum

    // the machine is repaired

    let! t = Dynamics.time |> Dynamics.lift
    let! dt = Parameter.dt |> Parameter.lift
    let! upTime =
        Parameter.randomExponential meanUpTime
            |> Parameter.lift

    startUpTime := t
    upNum := int (upTime / dt)
}

// return a simulation model of the machine
eventive {

    if !upNum > 0 then
```

```
                return! untilBroken
            elif !upNum = 0 then
                return! broken
            elif !repairNum > 0 then
                return! untilRepaired
            elif !repairNum = 0 then
                return! repaired
            else
                return! repaired
        }

    // create two machines

    let m1 = machine ()
    let m2 = machine ()

    // start the machines

    do! m1 |> Eventive.enqueueWithIntegTimes
           |> Eventive.runInStartTime

    do! m2 |> Eventive.enqueueWithIntegTimes
           |> Eventive.runInStartTime

    // return the result

    let upTimeProp =
        eventive {
            let! t = Dynamics.time |> Dynamics.lift
            return (!totalUpTime / (2.0 * t))
        }

    return
        [ResultSource.From ("upTimeProp", upTimeProp,
            "Long-run proportion of up time \
            (must be about 0.66)")]
                |> ResultSet.create
}
```

That was the model. Now we write the starting script in a separate file.

```
// File MachRep1ActivityOriented/Run.fsx

#load "Model.fsx"
```

```
open Model

open Simulation.Aivika
open Simulation.Aivika.Results

ResultSet.printInStopTime Model.specs Model.model
```

When running this script on macOS, we receive something similar to the following result.

```
bash-3.2$ fsharpi Run.fsx
----------

// time
t = 1000

// Long-run proportion of up time (must be about 0.66)
upTimeProp = 0,66335
```

We saw that the model written in this style is much longer. Nevertheless, the activity-oriented paradigm can be exceptionally useful for modeling some parts that are difficult to represent based on other simulation paradigms.

Chapter 3

Resources

This document illustrates how more and more high level concepts can be applied to modeling and simulation, when using IronAivika. The resources considered in this chapter are not exception. They are somewhere of intermediate level, but they do allow simplifying many models, nevertheless.

3.1 Queue Strategies

Before we proceed to more high level modeling constructs, we need to define the *queue strategies*[11] that prescribe how the competitive requests must be prioritized.

In IronAivika the queue strategies are expressed in terms of the auxiliray `IQueueStrategy` interface, where each queue strategy has its own implementation.

```
type Priority = float

[<Interface>]
type IQueueStorage<'a> =

    abstract IsEmpty : unit -> Eventive<bool>
    abstract Dequeue : unit -> Eventive<'a>
    abstract Enqueue : item:'a -> Eventive<unit>
    abstract Enqueue : priority:Priority * item:'a -> Eventive<unit>

[<Interface>]
```

```
type IQueueStrategy =
    abstract CreateStorage<'a> : unit -> Simulation<IQueueStorage<'a>>
```

The queue strategy must implement the `Dequeue` method and one of the `Enqueue` methods. Another method must raise an exception when it is called. The second `Enqueue` method is destined for strategies based on priorities, while the first one is implemented by more simple queue strategies.

There are four predefined queue strategies in IronAivika[1]:

- FCFS (First Come - First Served), or FIFO (First In - First Out);
- LCFS (Last Come - First Served), or LIFO (Last In - First Out);
- SIRO (Service in Random Order);
- StaticPriorities (Using Static Priorities), where the less value means a higher priority.

These strategies are implemented by the corresponded class types.

```
[<Sealed>]
type FCFS =
    interface IQueueStrategy

[<Sealed>]
type LCFS =
    interface IQueueStrategy

[<Sealed>]
type SIRO =
    interface IQueueStrategy

[<Sealed>]
type StaticPriorities =
    interface IQueueStrategy
```

There is also a module that contains predefined values of these types for convenience.

[1] Also the `Block` computations considered below introduce two additional strategies `TransactFCFS` and `TransactLCFS`.

```
module QueueStrategy =

    val FCFS : FCFS
    val LCFS : LCFS
    val SIRO : SIRO
    val staticPriorities : StaticPriorities
```

3.2 Resource

A *resource*[4] simulates something to be queued for, for example, the machine.

```
[<Sealed>]
type Resource
```

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

```
module Resource =
    val create: strat:#IQueueStrategy -> count:int -> Simulation<Resource>
```

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

```
module Resource =

    val request : Resource -> Proc<unit>
    val requestWithPriority : Priority -> Resource -> Proc<unit>
```

Each of the both suspends the process in case of the resource deficiency until some other simulation activity releases the resource.

```
module Resource =
    val releaseWithinEventive : Resource -> Eventive<unit>
```

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

```
module Resource =
    val release : Resource -> Proc<unit>
```

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

```
module Resource =

    val count : Resource -> Eventive<int>
    val maxCount : Resource -> int option
    val strategy : Resource -> IQueueStrategy
```

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

```
module Resource =
    val createWithMaxCount: strat:#IQueueStrategy
                            -> count:int
                            -> maxCount:int option
                            -> Simulation<Resource>
```

By default, the maximum possible amount is set equaled to the initial amount specified when calling the first constructor `create`.

There are constructors that use the predefined queue strategies. Some of these constructors are provided below.

```
module Resource =

    val createUsingFCFS : count:int -> Simulation<Resource>
    val createUsingLCFS : count:int -> Simulation<Resource>
    val createUsingSIRO : count:int -> Simulation<Resource>
    val createUsingPriorities : count:int -> Simulation<Resource>
```

Also there are two helper functions, where each of the both acquires the resource and returns `IDisposable` that in its turn allows releasing the resource regardless of whether the specified process was cancelled or an exception was raised. It can be used together with the `use!` construct of F#.

```
module Resource =

    val take : Resource -> Proc<IDisposable>
    val takeWithPriority : Priority -> Resource -> Proc<IDisposable>
```

Finally, we can increase the available amount of the resource to a new value, but not greater than the maximum amount that was defined at time of constructing the resource. Then some awaiting processes in the specified number can be awaken and they will acquire the resource.

```
module Resource =
    val incCount : n:int -> Resource -> Eventive<unit>
```

3.3 Example: Using Resources

To illustrates how the resources can be used for modeling, let us again take a task from the documentation of SimPy[9].

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

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

```
// File MachRep2/Model.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

open System

open Simulation.Aivika
open Simulation.Aivika.Results

let specs = {
```

```
    StartTime=0.0; StopTime=1000.0; DT=1.0;
    Method=RungeKutta4; GeneratorType=StrongGenerator
}

let meanUpTime = 1.0
let meanRepairTime = 0.5

let model = simulation {

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

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

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

    let! repairPerson = Resource.createUsingFCFS 1

    let machine = proc {

        while true do

            let! upTime = Proc.randomExponential meanUpTime
            totalUpTime := !totalUpTime + upTime

            incr nRep

            let! n =
                Resource.count repairPerson
                    |> Eventive.lift

            if n = 1 then
                incr nImmedRep

            do! Resource.request repairPerson
            let! repairTime = Proc.randomExponential meanRepairTime
            do! Resource.release repairPerson
    }

    do! Proc.runInStartTime machine
    do! Proc.runInStartTime machine
```

```
    let upTimeProp = eventive {

        let! t = Dynamics.time |> Dynamics.lift
        return !totalUpTime / (2.0 * t)
    }

    let immedTimeProp = eventive {
        return (float !nImmedRep) / (float !nRep)
    }

    return [ResultSource.From ("upTimeProp", upTimeProp,
                "Long-run proportion of up time \
                (must be about 0.6)");
            ResultSource.From ("immedTimeProp", immedTimeProp,
                "Long-run proportion of the time when \
                immediate access to the repairperson \
                (must be about 0.67)")]
                    |> ResultSet.create
}
```

Let us take the following simulation experiment: the specs, the deviation chart, statistics summary by last values, the histogram for last values. The number of simultaneous runs is 1000.

```
// File MachRep2/RunExperiment.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.dll"

#load "Model.fsx"

open Model

open System
open System.IO

open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Web
```

```
let experiment = Experiment ()

experiment.Specs <- Model.specs
experiment.RunCount <- 1000

let provider1 = ExperimentSpecsProvider ()
let provider2 = DeviationChartProvider ()
let provider3 = LastValueStatsProvider ()
let provider4 = LastValueHistogramProvider ()

let providers =
    [ provider1 :> IExperimentProvider<TextWriter>;
      provider2 :> IExperimentProvider<TextWriter>;
      provider3 :> IExperimentProvider<TextWriter>;
      provider4 :> IExperimentProvider<TextWriter> ]

experiment.RenderHtml (Model.model, providers)
    |> Async.RunSynchronously
```

The corresponding histogram is shown on figure 3.1.

3.4 Example: Passivating and Reactivating Processes

This example illustrates how we can passivate and reactivate processes depending on that whether the resource is free. The task corresponds to the documentation of SimPy[9].

> Variation of the previous models described in sections 2.6 and 3.3. 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.

In IronAivika the corresponding model can be defined in the following way.

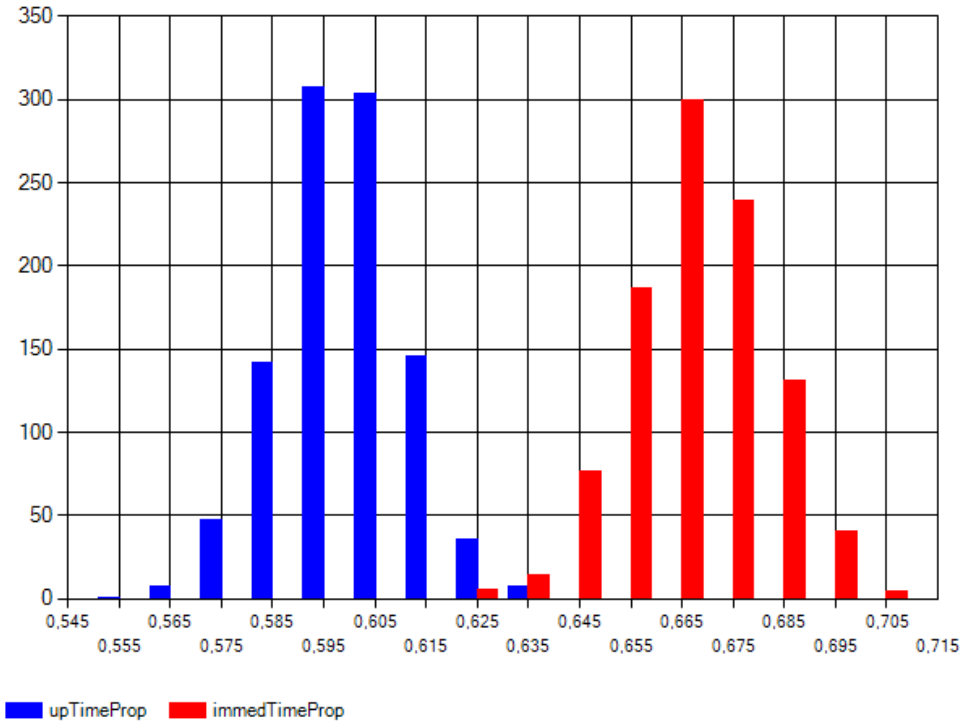


Figure 3.1: The histogram for time proportions, when using the resource.

```
// File MachRep3/Model.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

open Simulation.Aivika
open Simulation.Aivika.Results

let specs = {

    StartTime=0.0; StopTime=1000.0; DT=1.0;
    Method=RungeKutta4; GeneratorType=StrongGenerator
}

let meanUpTime = 1.0
let meanRepairTime = 0.5

let model = simulation {

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

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

    let! repairPerson = Resource.createUsingFCFS 1

    let machine pid' = proc {

        incr nUp

        while true do

            let! upTime = Proc.randomExponential meanUpTime
            totalUpTime := !totalUpTime + upTime

            decr nUp

            if !nUp = 1 then
                do! Proc.passivate
            else
                let! n =
                    Resource.count repairPerson
```

```
                        |> Eventive.lift

                if n = 1 then
                    do! Proc.reactivate pid'
                            |> Eventive.lift

            do! Resource.request repairPerson
            let! repairTime = Proc.randomExponential meanRepairTime

            incr nUp

            do! Resource.release repairPerson
    }

    let! pid1 = Proc.createId
    let! pid2 = Proc.createId

    do! Proc.runInStartTimeUsingId pid1 (machine pid2)
    do! Proc.runInStartTimeUsingId pid2 (machine pid1)

    let upTimeProp = eventive {
            let! t = Dynamics.time |> Dynamics.lift
            return (!totalUpTime / (2.0 * t))
    }

    return [ResultSource.From ("upTimeProp", upTimeProp,
                "The proportion of up time \
                (must be about 0.45)")]
                    |> ResultSet.create
}
```

Let us take the same simulation experiment that we used in section 3.3: show the specs, the deviation chart, statistics summary by last values, the histogram for last values. The number of simultaneous runs is 1000.

```
// File MachRep3/RunExperiment.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.dll"

#load "Model.fsx"
```

```
open Model

open System
open System.IO

open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Web

let experiment = Experiment ()

experiment.Specs <- Model.specs
experiment.RunCount <- 1000

let provider1 = ExperimentSpecsProvider ()
let provider2 = DeviationChartProvider ()
let provider3 = LastValueStatsProvider ()
let provider4 = LastValueHistogramProvider ()

let providers =
    [ provider1 :> IExperimentProvider<TextWriter>;
      provider2 :> IExperimentProvider<TextWriter>;
      provider3 :> IExperimentProvider<TextWriter>;
      provider4 :> IExperimentProvider<TextWriter> ]

experiment.RenderHtml (Model.model, providers)
    |> Async.RunSynchronously
```

The corresponding histogram is shown on figure 3.2. We expect to see a value distributed near 0.45.

3.5 Resource Preemption

There are simulation tasks that are modeled better with help of resource preemption. A preemptible resource is like the ordinary resource parameterized by the queue strategy based on static priorities. The numeric priorities are necessary to request for the both resources.

However, there is an important difference. When trying to acquire the ordinary resource, the process suspends in case of resource deficiency. In

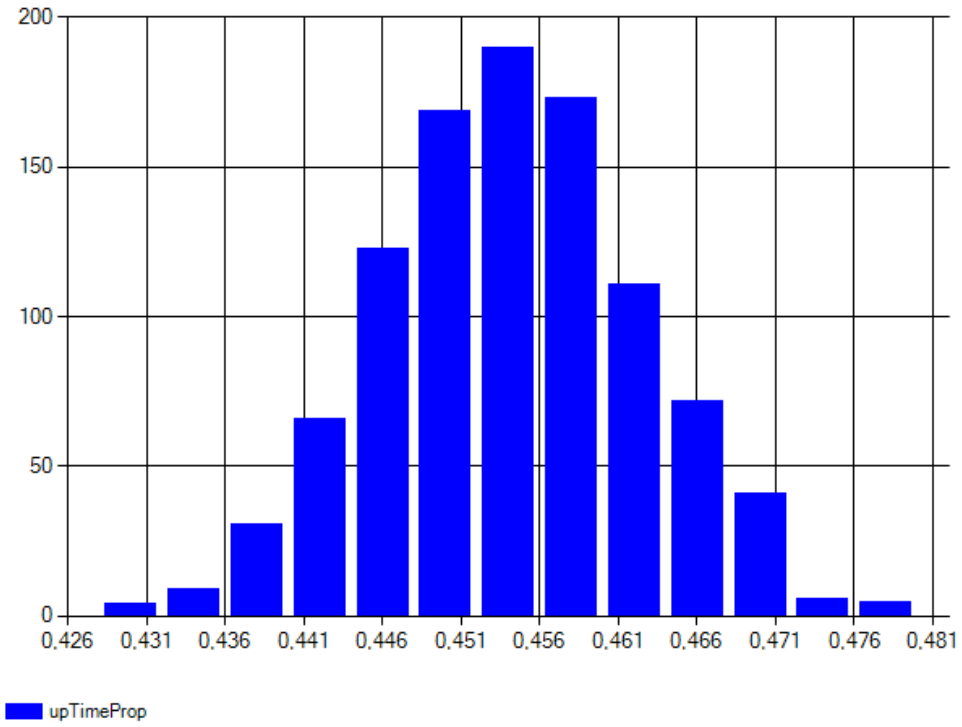


Figure 3.2: The histogram for time proportions, when passivating and reactivating the processes.

case of the resource preemption, the resource still can be acquired by the process even if the resource amount is zero, but then another process with less priority must be preempted and its ownership of the resource will be transferred to the current process with higher priority. If the current process that requests for the resource has a less priority then it waits for releasing of the resource, as usual.

So, the priority is used not only to range the process requests, but it allows also preempting another process transferring its ownership of the resource to another process.

```
[<Sealed>]
type PreemptibleResource

module PreemptibleResource =
```

```
    val count : PreemptibleResource -> Eventive<int>

    val maxCount : PreemptibleResource -> int option

    val create : count:int -> Simulation<PreemptibleResource>

    val createWithMaxCount : count:int
                             -> maxCount:int option
                             -> Simulation<PreemptibleResource>

    val requestWithPriority : Priority
                              -> PreemptibleResource
                              -> Proc<unit>

    val release : PreemptibleResource -> Proc<unit>

    val takeWithPriority : Priority
                           -> PreemptibleResource
                           -> Proc<IDisposable>
```

Note that here the resource must be released only within the corresponding `Proc` computation. The releasing operation has $O(n)$ complexity, where n is the number of the current owners, while the releasing of the ordinary resource is quite fast.

It is important to know what namely processes took the ownership of the resource. Having this information, we can preempt one of these processes if another process with higher priority requests for the resource.

As it was true with the static priority queue strategy, the less value means a higher priority.

Since the resources can be preempted at any time, we can either increase or decrease the available amount of the resource.

```
module PreemptibleResource =

    val incCount: n:int -> PreemptibleResource -> Eventive<unit>
    val decCount: n:int -> PreemptibleResource -> Eventive<unit>
    val alterCount: n:int -> PreemptibleResource -> Eventive<unit>
```

Section 6.9 illustrates an example of using the resource preemption.

Chapter 4

Signals and Tasks

The constructs considered in this chapter are closer to programming than to simulation. Nevertheless, they can be very useful for modeling.

4.1 Signals

A *signal* is a variation of the standard `IObservable` interface but specialized for modeling.

```
[<AbstractClass; NoEquality; NoComparison>]
type Signal<'a> =

    new: unit -> Signal<'a>

    abstract Subscribe: handler:('a -> Eventive<unit>)
                        -> Eventive<IDisposable>
```

The `Subscribe` method of the signal takes the handler, subscribes it for receiving the signal values and then returns a computation of the `IDisposable` object that, being invoked, unsubscribes the specified handler from receiving the signal.

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

```
[<AutoOpen>]
module SignalExtensions =

    type Signal<'a> with
        member Add: handler:('a -> Eventive<unit>) -> Eventive<unit>
```

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

```
module Signal =

    val add : ('a -> Eventive<unit>) -> Signal<'a> -> Eventive<unit>

    val subscribe: ('a -> Eventive<unit>)
                   -> Signal<'a>
                   -> Eventive<IDisposable>

    val map : ('a -> 'b) -> Signal<'a> -> Signal<'b>
    val filter : ('a -> bool) -> Signal<'a> -> Signal<'a>

    val empty<'a> : Signal<'a>
    val merge : Signal<'a> -> Signal<'a> -> Signal<'a>
    val concat : #Signal<'a> list -> Signal<'a>
```

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

```
 module Ref =

    val changed : Ref<'a> -> Signal<'a>
    val changed_ : Ref<'a> -> Signal<unit>

 module Var =

    val changed : Var<'a> -> Signal<'a>
    val changed_ : Var<'a> -> Signal<unit>
```

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

```
[<Sealed>]
type SignalSource<'a>

module SignalSource =

    val create<'a> : Simulation<SignalSource<'a>>

    val publish : source:SignalSource<'a> -> Signal<'a>
    val trigger : value:'a -> source:SignalSource<'a> -> Eventive<unit>
```

Moreover, we can register a history of signal values, which can be useful for accumulating the simulation results.

```
[<Sealed>]
type SignalHistory<'a>

module SignalHistory =

    val create : signal:Signal<'a> -> Eventive<SignalHistory<'a>>

    val read : history:SignalHistory<'a>
               -> Eventive<Time array * 'a array>
```

The `SignalHistory` type is widely used in the implementation of simulation experiments, when the results are accumulated by signals, for which there are predefined signals.

```
module Signal =

    val inTimes : times:#seq<Time> -> Eventive<Signal<Time>>
    val inIntegTimes : Eventive<Signal<Time>>
    val inStartTime : Eventive<Signal<Time>>
    val inStopTime : Eventive<Signal<Time>>
```

To plot the histogram, we accumulate the simulation results by the signal triggered in the final time, while we use the signal triggered in the integration time points to save the CSV table or draw the time series chart. Only we have to transform the source signal to receive the values we need.

Finally, you might notice that the `Cont` and `Signal` computations have a similar definition. This relation is expressed by the following function, where the current process suspends until the next signal value is triggered. Then the signal value is passed in to the process computation, which is resumed.

```
module Proc =
    val await : signal:Signal<'a> -> Proc<'a>
```

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

4.2 Tasks

A *task* encompasses the process computation started in background.

```
[<Sealed>]
type Task<'a>

module Task =
    val run : comp:Proc<'a> -> Eventive<Task<'a>>
```

Here we run the specified process in background and immediately return the corresponding task within the `Eventive` computation. Later, we can request for the result of the underlying `Proc` computation, whether it was finished successfully, or an exception occurred, or the computation was cancelled.

```
type TaskResult<'a> =
    | TaskCompleted of 'a
    | TaskError of exn
    | TaskCancelled

module Task =

    val tryGetResult : task:Task<'a> -> Eventive<TaskResult<'a> option>
    val result : task:Task<'a> -> Proc<TaskResult<'a>>
    val resultReceived : task:Task<'a> -> Signal<TaskResult<'a>>
```

The background task can be cancelled at any time.

```
module Task =

    val cancel : task:Task<'a> -> Eventive<unit>
    val isCancelled : task:Task<'a> -> Eventive<bool>
```

Also we can include the task computation into an arbitrary `Proc` computation, making the former a compound part of the latter.

```
module Task =
    val toProc : task:Task<'a> -> Proc<'a>
```

In some sense the `Signal` and `Task` types complement other simulation computations, which illustrates how deeply different computations can be inter-connected to each other, allowing us to define more comprehensive models.

Chapter 5

Statistics

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

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

5.1 Statistics based upon Observations

The generic `SamplingStats` data type is used for accumulating statistics based upon observations.

```
type SamplingStats<'a>

module SamplingStats =

    val emptyInts : SamplingStats<int>
    val emptyFloats : SamplingStats<float>

    val fromInts : samples:int array -> SamplingStats<int>
    val fromFloats : samples:float array -> SamplingStats<float>

    val add : sample:'a -> stats:SamplingStats<'a> -> SamplingStats<'a>

    val append : stats1:SamplingStats<'a>
                 -> stats2:SamplingStats<'a>
```

```
                -> SamplingStats<'a>

    val appendChoice : stats1:Choice<'a, SamplingStats<'a>>
                       -> stats2:SamplingStats<'a>
                       -> SamplingStats<'a>

    val appendSeq : samples:seq<'a>
                    -> stats:SamplingStats<'a>
                    -> SamplingStats<'a>

    val count : stats:SamplingStats<'a> -> int
    val minimum : stats:SamplingStats<'a> -> 'a
    val maximum : stats:SamplingStats<'a> -> 'a

    val mean : stats:SamplingStats<'a> -> float
    val mean2 : stats:SamplingStats<'a> -> float
    val variance : stats:SamplingStats<'a> -> float
    val deviation : stats:SamplingStats<'a> -> float

    val fromIntsToFloats : stats:SamplingStats<int>
                           -> SamplingStats<float>
```

The main idea is that this is an immutable data type. Each time we collect a new sample, we actually create a new instance of the `SamplingStats` type.

It would be a mistake to use rather a heavy-weight `Var` type for collecting the statistics only. Nevertheless, it is recommended to use the standard `ref` reference or predefined `Ref` type for updating the light-weight `SamplingStats` value, which is a more efficient and more simple approach.

```
let r = ref SamplingStats.emptyFloats
...

r := !r |> SamplingStats.add 1.0
r := !r |> SamplingStats.add 2.0
...
```

The `SamplingStats` values can be returned as a `ResultSource` within the `Simulation` computation.

5.2 Statistics for Time Persistent Variables

The generic `TimingStats` data type is used for collecting time-dependent statistics.

```
type TimingStats<'a>

module TimingStats =

    val emptyInts : TimingStats<int>
    val emptyFloats : TimingStats<float>

    val add : time:Time
              -> sample:'a
              -> stats:TimingStats<'a>
              -> TimingStats<'a>

    val count : stats:TimingStats<'a> -> int
    val minimum : stats:TimingStats<'a> -> 'a
    val maximum : stats:TimingStats<'a> -> 'a
    val last : stats:TimingStats<'a> -> 'a

    val minimumTime : stats:TimingStats<'a> -> Time
    val maximumTime : stats:TimingStats<'a> -> Time
    val startTime : stats:TimingStats<'a> -> Time
    val lastTime : stats:TimingStats<'a> -> Time

    val sum : stats:TimingStats<'a> -> float
    val sum2 : stats:TimingStats<'a> -> float
    val mean : stats:TimingStats<'a> -> float
    val mean2 : stats:TimingStats<'a> -> float
    val variance : stats:TimingStats<'a> -> float
    val deviation : stats:TimingStats<'a> -> float

    val fromIntsToFloats : stats:TimingStats<int> -> TimingStats<float>
```

The `TimingStats` value is immutable too. As before, it can be used when it is wrapped in the mutable reference.

There is also one function that allows converting the `TimingStats` statistics to its normalized representation based upon observations. We interpolate the former so that it would be statistically similar to the latter by the specified number of pseudo-observations.

```
module TimingStats =
    val normalise: count:int -> stats:TimingStats<'a> -> SamplingStats<'a>
```

For example, this function is used when plotting the deviation chart for queue sizes. These sizes are time persistent variables, while the deviation chart plots the trend and confidence intervals for the statistics based upon observations. Normalizing the queue size statistics by the iteration number, we receive another representation of the queue size by which we can plot the deviation chart.

As before, the `TimingStats` values can be returned as a `ResultSource` within the `Simulation` computation.

Chapter 6

Queue Network

It is difficult to imagine any complex discrete event simulation without using queues. This chapter introduces the queues and shows how we can model networks based on them.

However, the most part of this chapter is considered to be obsolete and outdated except mostly for the `Stream` computation. Instead, it is highly recommended to consider using the `Block` computations described further. By the way, these blocks use the streams too. So, you can skip the most part of this chapter except for those sections which describe the streams and experiment providers. The streams are used for creating transacts, which are then processed by the `Block` computations. The experiment providers are used for returning the results of simulation.

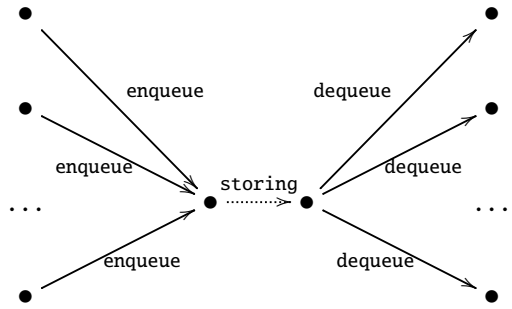
6.1 Bounded Queues

Sometimes we need a location in the network where entities wait for service[5]. They are modeled in IronAivika by finite and infinite queues.

The finite (bounded) queue is a container of elements.

```
[<Sealed>]
type Queue<'a>
```

To create a new queue, we should specify the queue strategies that will be used for ranging the enqueueing operations, internal storing of items in the queue and ranging the dequeueing operations respectively. Also we should specify the queue capacity, for this queue is finite.



The enqueueing strategy is used for ranging the enqueueing operations when the queue is full. The storing strategy is used for ranging the items in the queue itself. The dequeueing strategy is used for ranging the dequeueing requests when the queue is empty.

The first and third strategies used for ranging the enqueueing and dequeueing operations usually should be defined as the FCFS strategy, i.e. first come - first serviced, which is the most intuitive and which suits the most of needs, while the storing strategy distinguishes the queue itself.

In general case, the queue constructor is as follows.

```
module Queue =

    val create<'si, 'sm, 'so, 'a
                   when 'si :> IQueueStrategy and
                        'sm :> IQueueStrategy and
                        'so :> IQueueStrategy> :
                           inputStrat:'si
                           -> storingStrat:'sm
                           -> outputStrat:'so
                           -> maxCount:int
                           -> Eventive<Queue<'a>>
```

Fortunately, there are specializations that allow creating new queues using the predefined strategies and these specializations look much shorter.

```
module Queue =

    val createUsingFCFS<'a> : maxCount:int -> Eventive<Queue<'a>>
    val createUsingLCFS<'a> : maxCount:int -> Eventive<Queue<'a>>
    val createUsingSIRO<'a> : maxCount:int -> Eventive<Queue<'a>>
    val createUsingPriorities<'a> : maxCount:int -> Eventive<Queue<'a>>
```

Each of them uses the FCFS strategy for ranging the enqueueing and dequeueing operations, but uses the corresponding queue strategy for the internal storing.

For example, the createUsingFCFS function uses FCFS for the storing operation too, while the createUsingPriorities function creates already a queue that uses the static priorities, when storing a new element.

Unlike other data structures, a queue is created within the Eventive computation as we have to know the current simulation time to start gathering the timing statistics for the queue size. The statistics is initiated at time of invoking the computation.

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

```
module Queue =
    val enqueue : item:'a -> queue:Queue<'a> -> Proc<unit>
```

It suspends the process if the finite queue is full. Therefore, this action is returned as the Proc computation.

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

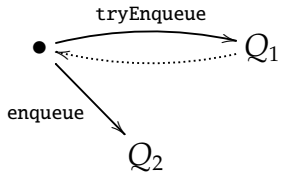
```
module Queue =
    val enqueueOrLost : item:'a -> queue:Queue<'a> -> Eventive<bool>
```

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

There is also a similar function that tries to enqueue the new item, but in case of the full queue the item is not counted as lost and the queue statistics does not change.

```
module Queue =
    val tryEnqueue : item:'a -> queue:Queue<'a> -> Eventive<bool>
```

This function can be useful if we are going to enqueue the item in another queue in case of failure. We try the first queue Q_1, fail and then enqueue the item in the second queue Q_2.



If the queue was created by applying the `createUsingPriorities` function then we must enqueue a new element specifying also the storing priority.

```
module Queue =

    val enqueueWithStoringPriority : pm:Priority
                                     -> item:'a
                                     -> queue:Queue<'a>
                                     -> Proc<unit>

    val enqueueWithStoringPriorityOrLost : pm:Priority
                                           -> item:'a
                                           -> queue:Queue<'a>
                                           -> Eventive<bool>

    val tryEnqueueWithStoringPriority : pm:Priority
                                        -> item:'a
                                        -> queue:Queue<'a>
                                        -> Eventive<bool>
```

There are also other enqueueing functions that allow us to specify the priority to be used when ranging the enqueueing operations in case of full queue, but these functions are needed only if we specified the corresponding priority-based enqueueing strategy, when constructing the queue.

As it was mentioned before, the predefined queue constructors use the `FCFS` strategy for the enqueueing operation that needs no auxiliary priority, when ranging the operations if the queue is full. The first operation will have a priority.

The simplest dequeueing operation suspends the process while the queue is empty. The result is the `Proc` computation.

```
module Queue =
    val dequeue : queue:Queue<'a> -> Proc<'a>
```

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

There are similar dequeueing functions that allow us to specify the priority to be used when ranging the dequeueing operations if the queue is empty. As before, these similar functions are needed only if the queue was constructed by specifying the priority-based dequeueing strategy.

Regarding the predefined queue constructors, they also use the `FCFS` strategy for ranging the dequeueing operations, which needs no auxiliary priority. The first operation will have a priority if the queue is empty.

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

For example, we can request the queue for its size and wait time.

```
module Queue =

    val countStats : Queue<'a> -> Eventive<TimingStats<int>>
    val waitTime : Queue<'a> -> Eventive<SamplingStats<Time>>
```

Finally, we can return an arbitrary queue or a list of queues from the model as a `ResultSource`. Then the queue or the list of queues can be used within the simulation experiment.

6.2 Unbounded Queues

The infinite (unbounded) queue is a container of elements.

```
[<Sealed>]
type InfiniteQueue<'a>
```

Since the queue is unbounded, there is no need to range the enqueueing operations as the queue cannot be full. Therefore, a new queue is created by specifying the storing and dequeueing strategies respectively.

```
module InfiniteQueue =

    val create<'sm, 'so, 'a
                   when 'sm :> IQueueStrategy and
                        'so :> IQueueStrategy> :
                          storingStrat:'sm
                          -> outputStrat:'so
                          -> Eventive<InfiniteQueue<'a>>
```

The storing strategy is used for ranging the elements in the queue itself. The dequeueing strategy is used for ranging the dequeueing operations if the queue is empty.

There are specializations that allow creating new queues based on the predefined queue strategies.

```
module InfiniteQueue =

    val createUsingFCFS<'a> : Eventive<InfiniteQueue<'a>>
    val createUsingLCFS<'a> : Eventive<InfiniteQueue<'a>>
    val createUsingSIRO<'a> : Eventive<InfiniteQueue<'a>>
    val createUsingPriorities<'a> : Eventive<InfiniteQueue<'a>>
```

Each of them uses the `FCFS` strategy, i.e. first come - first serviced, for ranging the dequeueing operations in case of empty queue. The first operation will have a priority.

There are only two enqueueing functions, and their actions are performed immediately without suspension. The simulation time remains the same after the operation.

```
module InfiniteQueue =

    val enqueue : item:'a -> queue:InfiniteQueue<'a> -> Eventive<unit>

    val enqueueWithStoringPriority : pm:Priority
                                     -> item:'a
                                     -> queue:InfiniteQueue<'a>
                                     -> Eventive<unit>
```

The second function is used when the queue was created by specifying the priority-based storing strategy. In other cases the first function is used.

The simplest dequeueing operation suspends the process while the queue is empty. The result is the `Proc` computation.

```
module InfiniteQueue =
    val dequeue : queue:InfiniteQueue<'a> -> Proc<'a>
```

It is worth noting again that the type signatures specify whether the corresponding function may suspend the simulation activity, or the action is performed immediately.

There is a similar dequeueing function that allows us to specify the priority to be used when ranging the dequeueing operations if the queue

is empty. As before, this similar function is needed only if the queue was constructed by specifying the priority-based dequeueing strategy.

Regarding the predefined queue constructors, they use the FCFS strategy for ranging the dequeueing operations, which needs no auxiliary priority. The first operation will have a priority if the unbounded queue is empty.

Like the bounded queue, the infinite queue has a lot of counters which are updated during the simulation.

For example, we can request the infinite queue for its size and wait time.

```
module InfiniteQueue =

    val countStats : InfiniteQueue<'a> -> Eventive<TimingStats<int>>
    val waitTime : InfiniteQueue<'a> -> Eventive<SamplingStats<Time>>
```

Also we can return an arbitrary infinite queue or a list of such queues from the model as a `ResultSource`. Then the infinite queue or the list of queues can be used within the simulation experiment.

6.3 Stream

Many things become significantly more simple for reasoning and understanding after we introduce a concept of *stream* of data distributed sequentially in the modeling time.

```
type StreamItem<'a> =
    | StreamNil
    | StreamCons of 'a * Stream<'a>

and Stream<'a> = Stream of Proc<StreamItem<'a>>
```

The corresponding computation builder has name `stream`:

```
let x : Stream<'a> = stream { .. }
```

It supports F# keywords `yield`, `yield!` in obvious way. Also it supports the `let!` and `do!` constructs allowing us to embed arbitrary `Proc` computation in the `Stream` computation.

```
let x : Stream<'a> = stream {
        let a : 'a = ...
        yield a
        ...
        let p : Proc<'b> = ...
        let! b : 'b = p
        ...
    }
```

The `Stream` type is a kind of the *cons-cell*, where the cell is returned within the `Proc` computation. It means that the stream data can be distributed in the modeling time and there can be time gaps between sequential data.

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

```
module Stream =

    val map : ('a -> 'b) -> Stream<'a> -> Stream<'b>
    val mapc : ('a -> Proc<'b>) -> Stream<'a> -> Stream<'b>

    val filter : ('a -> bool) -> Stream<'a> -> Stream<'a>
    val filterc : ('a -> Proc<bool>) -> Stream<'a> -> Stream<'a>
```

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

```
module Stream =
    val empty<'a> : Stream<'a>
```

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

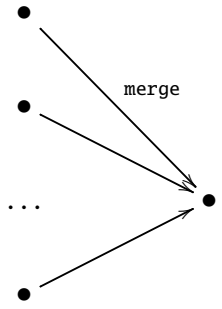
```
module Stream =
    val append : Stream<'a> -> Stream<'a> -> Stream<'a>
```

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

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

```
module Stream =

    val merge : Stream<'a> list -> Stream<'a>
    val mergeQueueing : #IQueueStrategy -> Stream<'a> list -> Stream<'a>
    val mergePrioritising : #IQueueStrategy
                            -> Stream<Priority * 'a> list
                            -> Stream<'a>
```

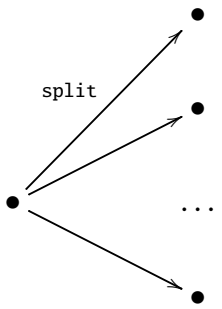
The functions use the resources to concatenate different streams of data.



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

```
module Stream =

    val split : int -> Stream<'a> -> Stream<'a> list
    val splitQueueing : #IQueueStrategy
                        -> int
                        -> Stream<'a>
                        -> Stream<'a> list
    val splitPrioritising : #IQueueStrategy
                            -> Stream<Priority> list
                            -> Stream<'a>
                            -> Stream<'a> list
```

These functions use also the resources to split the stream.



The implementation uses an auxiliary function that creates a new stream as the result of repetitive execution of some process.

```
module Stream =
    val repeat : Proc<'a> -> Stream<'a>
```

The key idea is that many simulation models can be defined as a network of `Stream` computations[2].

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

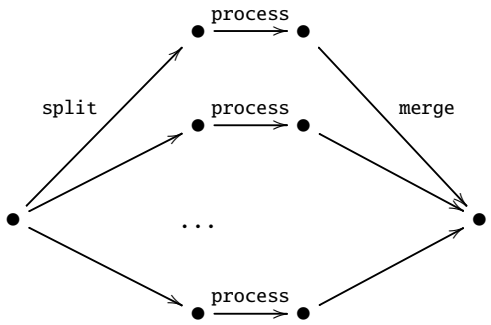
```
module Stream =

    val randomUniform : minimum:float
                        -> maximum:float
                        -> Stream<Arrival<float>>

    val randomUniformInt : minimum:int
                           -> maximum:int
                           -> Stream<Arrival<int>>

    val randomNormal : mean:float
                       -> deviation:float
                       -> Stream<Arrival<float>>

    val randomExponential : mean:float -> Stream<Arrival<float>>
    val randomErlang : beta:float -> m:int -> Stream<Arrival<float>>
    val randomPoisson : mean:float -> Stream<Arrival<int>>
    val randomBinomial : prob:float -> trials:int -> Stream<Arrival<int>>
```

[2]It was too naive. The `Block` computations suit this task much better, but they use the `Stream` computation too.

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

```
type Arrival<'a> =
    { Value : 'a;
      Time : float;
      Delay : float option }
```

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



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

```
module Stream =
    val prefetch : Stream<'a> -> Stream<'a>
```

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

```
module Stream =
    val sink : Stream<'a> -> Proc<unit>
```

It infinitely reads data from the specified stream.

When building queue networks, the following function can be useful too.

```
module Stream =
    val memo : Stream<'a> -> Stream<'a>
```

It memoizes the stream so that the resulting stream would always return the same data within the simulation run.

> Below is described the section that elaborates further the idea of using `Stream` computations to define the queue networks. If you will find this material difficult to understand, then you are not alone. I also consider them difficult for use, even although I am the author. At time present, I find that the `Block` computations are much easier to understand and to use, they are more safe and robust. But they apply the `Stream` computation as a source of arrival events, from which the transacts are created to be processed then by blocks. Nevertheless, I decided to retain the extended study material related to the `Stream` computation for more deep understanding.

6.4 Processor

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

```
type Processor<'a, 'b> = Stream<'a> -> Stream<'b>
```

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

A new processor can be created by the specified handling function that produces the output that can be either pure or the `Proc` computation.

```
module Processor =

    val arr : ('a -> 'b) -> Processor<'a, 'b>
    val arrc : ('a -> Proc<'b>) -> Processor<'a, 'b>
```

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

```
module Processor =
    val accum : ('st -> 'a -> Proc<'st * 'b>) -> 'st -> Processor<'a, 'b>
```

We can involve the `Proc` computation with side effect, when processing every element of the input stream of data.

```
module Processor =
    val within : Proc<unit> -> Processor<'a, 'a>
```

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

```
module Processor =
    val par : Processor<'a, 'b> list -> Processor<'a, 'b>
```

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

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

To create a sequence of autonomously working processors, we can use the next function, which is based on the prefetching function for streams considered above too:

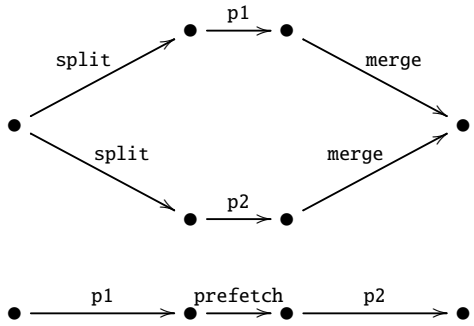
```
module Processor =
    val seq : Processor<'a, 'a> list -> Processor<'a, 'a>
```

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

```
let pPar = Processor.par [p1; p2]
let pSeq = Processor.seq [p1; p2]
```

The latter could be written explicitly as

```
let pSeq = p1 >> Stream.prefetch >> p2
```



We could connect two processors `p1` and `p2` directly, but it would be a monolithic processor, where the input element is requested only after the output element is requested outside.

```
let pWhole = p1 >> p2
```

When creating a sequence of processors, we have to isolate the processors with help of intermediate buffers, where the `Stream.prefetch` processor plays a role of such an active buffer that requests one more element in advance.

Table 6.1: Composing Processors.

Function	Description
`Processor.par`	Parallel processors
`Processor.seq`	Sequential processors
`(»)`	Processor composition

By the same reason, we can connect directly to the queue processor as the queue is also an example of active buffer.

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

```
module Processor =

    val queue : enqueue:('a -> Proc<unit>)
                -> dequeue:Proc<'b>
                -> Processor<'a, 'b>
```

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

Therefore, it was decided to introduce such general-purpose helper functions for modeling the queues, where the details of how the queues are simulated can be shortly described with help of combinators like `enqueue`, `enqueueOrLost` or `dequeue`.

However, there are three predefined combinators that cover the most of cases. Each of them returns a processor corresponding to the queue and implements the desired behavior.

```
module Queue =

    val processor : Queue<'a> -> Processor<'a, 'a>
    val processorWithLost : Queue<'a> -> Processor<'a, 'a>

module InfiniteQueue =

    val processor : InfiniteQueue<'a> -> Processor<'a, 'a>
```

The definition of these three queue processors is quite simple. We pass in two computations to the `queue` combinator. The first computation defines how the input elements are enqueued. The second computation defines how the output elements are dequeued.

```
module Queue =

    let processor (queue: Queue<'a>) =
            Processor.queue
                (fun a -> queue |> Queue.enqueue a)
                (Queue.dequeue queue)

    let processorWithLost (queue:Queue<'a>) =
            Processor.queue
                (fun a -> queue |> Queue.enqueueOrLost_ a
                                |> Eventive.lift)
                (Queue.dequeue queue)

module InfiniteQueue =
```

```
    let processor (queue: InfiniteQueue<'a>) =
            Processor.queue
                (fun a -> queue |> InfiniteQueue.enqueue a
                                |> Eventive.lift)
                (InfiniteQueue.dequeue queue)
```

Also we can model the queue networks with loopbacks by using the intermediate queues to delay the stream. One of the possible combinators is provided below.

```
module Processor =

    val queueLoopSeq : enqueue:('a -> Proc<unit>)
                       -> dequeue:Proc<'c>
                       -> cond:Processor<'c, Choice<'e, 'b>>
                       -> body:Processor<'e, 'a>
                       -> Processor<'a, 'b>
```

Moreover, there are helper functions that hold the active process for a random time interval according to the desired distribution.

```
module Processor =

    val randomUniform : minimum:float
                        -> maximum:float
                        -> Processor<'a, 'a>

    val randomUniformInt : minimum:int
                           -> maximum:int
                           -> Processor<'a, 'a>

    val randomNormal : mean:float -> deviation:float -> Processor<'a, 'a>
    val randomExponential : mean:float -> Processor<'a, 'a>
    val randomErlang : beta:float -> m:int -> Processor<'a, 'a>
    val randomPoisson : mean:float -> Processor<'a, 'a>
    val randomBinomial : prob:float -> trials:int -> Processor<'a, 'a>
```

There is no magic in these random processors. Their definition is quite simple too:

```
module Processor =

    let randomUniform minimum maximum =
        Proc.randomUniform_ minimum maximum |> within
```

The example that would use the streams and queue processors is provided further in section 6.8.

Using the processors, we can model a complicated enough behavior, for example, we can model the Round-Robbin strategy[11] of the processing.

```
module Processor =
    val roundRobbin : Processor<Proc<Time> * Proc<'a>, 'a>
```

It tries to perform a task within the specified timeout. If the task times out, then it is canceled and returned to the processor again; otherwise, the successful result is redirected to output. The timeout and task are passed in to the processor from the input stream.

All computations are integrated in IronAivika, and we can combine different approaches within the same model, for example, by transforming an arbitrary `Dynamics` computation such as the integral to the high-level `Proc` computation and then by using it in the `Processor` computation.

> Unfortunately, it is quite difficult to use soley the `Stream` and based on them `Processor` computations to create queue networks. The streams are very useful as a source of arrival events, which can be processed then by more handy `Block` computations described further. VisualAivika uses namely the `Block` computations and some basic functions of the `Stream` computation.

6.5 Server

In IronAivika there is a `Server` data type that allows us to model a statefull working place and gathering its statistics[3].

```
[<Sealed>]
type Server<'state, 'a, 'b>

module Server =

    val create : f:('a -> Proc<'b>) -> Simulation<Server<unit, 'a, 'b>>

    val createAccum : f:('state -> 'a -> Proc<'state * 'b>)
                      -> state:'state
                      -> Simulation<Server<'state, 'a, 'b>>
```

[3]The `Server` data type is considered to be obsolete and outdated too.

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

By default, the server does not take into account a possible resource preemption, because its handling is quite a costly operation. Therefore, you should use more general constructors to create a server that would be able to properly gather its statistics in case of possible resource preemption.

```
module Server =

    val createPreemptible : preemptible:bool
                          -> f:('a -> Proc<'b>)
                          -> Simulation<Server<unit, 'a, 'b>>

    val createAccumPreemptible : preemptible:bool
                               -> f:('state -> 'a -> Proc<'state * 'b>)
                               -> state:'state
                               -> Simulation<Server<'state, 'a, 'b>>
```

Here the first argument defines whether the underlying process can be preempted, when acquiring the `PreemptibleResource` resource.

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

```
module Server =
    val processor : Server<'state, 'a, 'b> -> Processor<'a, 'b>
```

For example, we can request for the statistics of time spent by the server while processing the tasks.

```
module Server =
    val processingTime: Server<'state, 'a, 'b>
                      -> Eventive<SamplingStats<float>>
```

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

There are predefined servers that model an activity that holds the underlying process for a random time interval, when processing every input element.

```
module Server =

    val createRandomUniformPreemptible :
            preemptible:bool
            -> minimum:float
            -> maximum:float
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomUniform :
            minimum:float
            -> maximum:float
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomUniformIntPreemptible :
            preemptible:bool
            -> minimum:int
            -> maximum:int
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomUniformInt :
            minimum:int
            -> maximum:int
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomNormalPreemptible :
            preemptible:bool
            -> mean:float
            -> deviation:float
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomNormal :
            mean:float
            -> deviation:float
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomExponentialPreemptible :
            preemptible:bool
            -> mean:float
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomExponential :
```

```
            mean:float
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomErlangPreemptible :
            preemptible:bool
            -> beta:float
            -> m:int
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomErlang :
            beta:float
            -> m:int
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomPoissonPreemptible :
            preemptible:bool
            -> mean:float
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomPoisson :
            mean:float
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomBinomialPreemptible :
            preemptible:bool
            -> prob:float
            -> trials:int
            -> Simulation<Server<unit, 'a, 'a>>

    val createRandomBinomial :
            prob:float
            -> trials:int
            -> Simulation<Server<unit, 'a, 'a>>
```

The definition of these predefined servers with random activity is quite simple, which demonstrates how you can define your own activity.

```
module Server =

    let createRandomUniformPreemptible preemptible minimum maximum =
        createPreemptible preemptible (fun a -> proc {
            do! Proc.randomUniform_ minimum maximum
            return a
        })
```

```
let createRandomUniform minimum maximum =
    createRandomUniformPreemptible false minimum maximum
```

There is no difference between the predefined servers and custom-made ones. Moreover, the resource preemption will work even in case if the activity is defined as a complicated `Proc` computation with branches, calculations and so on.

6.6 Timing Arrivals

Usually, the input of the queue network is expressed as a random stream of `Arrival` values. While processing, we can modify data that come with the arrival. We can add new attributes, remove them and change. It is quite simple as the processors usually work with generic data. In the end, we have to measure the time which the specified arrival spent during the processing.

We can do this by using the following data type[4].

```
[<Sealed>]
type ArrivalTimer

module ArrivalTimer =
    val create : Simulation<ArrivalTimer>
```

The idea is that we pass the input data through a special processor that counts the time spent by arrivals.

```
module ArrivalTimer =
    val processor : ArrivalTimer -> Processor<Arrival<'a>, Arrival<'a>>
```

Then we request the timer for the processing time statistics collected.

```
module ArrivalTimer =
    val processingTime : ArrivalTimer -> Eventive<SamplingStats<float>>
```

Finally, the arrival timer can be returned as a `ResultSource` from the model.

[4]The `Block` computations do not use this data type.

6.7 Experiment Providers

The introduced above compound simulation entities such as queues and servers have a lot of counters. Either the compound entitity or its specific counter can be returned from the model as a `ResultSource`.

For example, to show the deviation chart and summary for the size statistics of the queue with name `queue1`, we could write:

```
let series : ResultTransform =
  ResultSet.findByName "queue1" >>
  ResultSet.findById QueueCountStatsId

let provider1 = DeviationChartProvider ()
let provider2 = LastValueStatsProvider ()

provider1.Series <- series
provider2.Series <- series
```

The pros of this approach is that the queue has many counters and we can specify precisely those ones that we need. Also we can specify how we want to display the simulation results. But such an approach seems to be quite tedious in practice as we usually need a small set of counters for quick analysis, for example, when validating the model.

Therefore, there is another way. The idea is that we can display only the most important information about the simulation entities by writing rather small code, where we just define what entity we want to display the information about.

At first, the basic experiment providers are redefined as easy-to-use combinators:

```
namespace Simulation.Aivika.Experiments.Web

module ExperimentProvider =

    /// Shows the experiment specs.
    val experimentSpecs: IExperimentProvider<TextWriter>

    /// Shows the information about the specified series.
    val description: series:ResultTransform
                     -> IExperimentProvider<TextWriter>

    /// Renders the last values for the specified series.
```

```
    val lastValue: series:ResultTransform
                   -> IExperimentProvider<TextWriter>

    /// Renders the CSV file with results for the specified series.
    val table: series:ResultTransform
               -> IExperimentProvider<TextWriter>

    /// Renders the CSV file with last results for the specified series.
    val lastValueTable: series:ResultTransform
                        -> IExperimentProvider<TextWriter>

    /// Renders the last value statistics for the specified series.
    val lastValueStats: series:ResultTransform
                        -> IExperimentProvider<TextWriter>
```

Similar combinators are defined at level of the charting component for fast creation of the predefined simulation providers.

```
namespace Simulation.Aivika.Charting.Web

module ExperimentProvider =

    /// Renders the time series for the specified series.
    val timeSeries: series:ResultTransform
                    -> IExperimentProvider<TextWriter>

    /// Renders the XY Chart for the specified series.
    val xyChart: seriesX:ResultTransform
                 -> seriesY:ResultTransform
                 -> IExperimentProvider<TextWriter>

    /// Renders the deviation chart for the specified series.
    val deviationChart: series:ResultTransform
                        -> IExperimentProvider<TextWriter>

    /// Renders the last value histogram for the specified series.
    val lastValueHistogram: series:ResultTransform
                            -> IExperimentProvider<TextWriter>
```

Now we use the property of providers, where they can be combined. We can take the list of providers and create a new provider that would behave as the whole.

```
namespace Simulation.Aivika.Experiments
```

```
module ExperimentProvider =

    /// Appends two providers.
    val append: p1:IExperimentProvider<'a>
                -> p2:IExperimentProvider<'a>
                -> IExperimentProvider<'a>

    /// Concatenates the specified providers.
    val concat: ps:IExperimentProvider<'a> list -> IExperimentProvider<'a>

    /// A provider that renders nothing.
    val empty<'a> : IExperimentProvider<'a>
```

Using this property, we can create easy-to-use simulation providers for some compound simulation entities by displaying only the most important information.

```
namespace Simulation.Aivika.Charting.Web

module ExperimentProvider =

    /// Renders the basic queue properties for the specified series.
    val queue: series:ResultTransform
               -> IExperimentProvider<TextWriter>

    /// Renders the basic queue properties for the specified series.
    val infiniteQueue: series:ResultTransform
                       -> IExperimentProvider<TextWriter>

    /// Renders the basic server properties for the specified series.
    val server: series:ResultTransform
                -> IExperimentProvider<TextWriter>

    /// Renders the basic arrival timer properties for
    /// the specified series.
    val arrivalTimer: series:ResultTransform
                      -> IExperimentProvider<TextWriter>
```

For example, the queue provider can be defined in the following way.

```
module ExperimentProvider =

    let queue (series: ResultTransform) =
        let series1 = series >> ResultSet.findById QueueCountStatsId
```

```
let series2 = series >> ResultSet.findById QueueWaitTimeId
let series3 = series >> ResultSet.findById QueueLostCountId
let series' = ResultTransform.concat [series; series1;
                                      series2; series3]
[ExperimentProvider.description series';
 deviationChart series1;
 ExperimentProvider.lastValueStats series1;
 deviationChart series2;
 ExperimentProvider.lastValueStats series2;
 deviationChart series3;
 ExperimentProvider.lastValueStats series3;
 lastValueHistogram series3]
    |> ExperimentProvider.concat
```

We see that this provider displays the information about the size statistics, wait time and the count of lost items for the queues specified.

6.8 Example: Work Stations in Series

To illustrate how the streams and processors can be used for modeling, let us consider a model [5, 11] of two work stations connected in a series and separated by finite queues.

> The maintenance facility of a large manufacturer performs two operations. These operations must be performed in series; operation 2 always follows operation 1. The units that are maintained are bulky, and space is available for only eight units including the units being worked on. A proposed design leaves space for two units between the work stations, and space for four units before work station 1. [..] Current company policy is to subcontract the maintenance of a unit if it cannot gain access to the in-house facility.
>
> Historical data indicates that the time interval between requests for maintenance is exponentially distributed with a mean of 0.4 time units. Service times are also exponentially distributed with the first station requiring on the average 0.25 time units and the second station, 0.5 time units. Units are transported automatically from work station 1 to work station 2 in a negligible amount of time. If the queue of work station 2 is full, that is, if

there are two units awaiting for work station 2, the first station is blocked and a unit cannot leave the station. A blocked work station cannot server other units.

Below is provided the corresponding simulation model.

```
// File WorkStationsInSeries/Model.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

open Simulation.Aivika
open Simulation.Aivika.Queues
open Simulation.Aivika.Results

/// the simulation specs
let specs = {

    StartTime=0.0; StopTime=300.0; DT=0.1;
    Method=RungeKutta4; GeneratorType=StrongGenerator
}

/// the mean delay of the input arrivals distributed exponentially
let meanOrderDelay = 0.4

/// the capacity of the queue before the first work places
let queueMaxCount1 = 4

/// the capacity of the queue before the second work places
let queueMaxCount2 = 2

/// the mean processing time distributed exponentially in
/// the first work stations
let meanProcessingTime1 = 0.25

/// the mean processing time distributed exponentially in
/// the second work stations
let meanProcessingTime2 = 0.5

/// the simulation model
let model = simulation {

    // it will gather the statistics about the processing time
```

```
let! arrivalTimer = ArrivalTimer.create

// define a stream of input events
let inputStream = Stream.randomExponential meanOrderDelay

// create a queue in front of the first work stations
let! queue1 =
    Queue.createUsingFCFS queueMaxCount1
        |> Eventive.runInStartTime

// create a queue between the first and second work stations
let! queue2 =
    Queue.createUsingFCFS queueMaxCount2
        |> Eventive.runInStartTime

// create the first work station (server)
let! workStation1 =
    Server.createRandomExponential meanProcessingTime1

// create the second work station (server)
let! workStation2 =
    Server.createRandomExponential meanProcessingTime2

// the entire processor from input to output
let entireProcessor =
    Queue.processorWithLost queue1 >>
    Server.processor workStation1 >>
    Queue.processor queue2 >>
    Server.processor workStation2 >>
    ArrivalTimer.processor arrivalTimer

// start simulating the model
do! inputStream
        |> entireProcessor
        |> Stream.sink
        |> Proc.runInStartTime

// return the simulation results
return [ResultSource.From ("queue1", queue1,
            "Queue no. 1");
        ResultSource.From ("workStation1", workStation1,
            "Work Station no. 1");
        ResultSource.From ("queue2", queue2,
            "Queue no. 2");
        ResultSource.From ("workStation2", workStation2,
```

```
                "Work Station no. 2");
            ResultSource.From ("arrivalTimer", arrivalTimer,
                "The arrival timer")]
                    |> ResultSet.create
}

/// the model summary
let modelSummary =
    model |> Simulation.map ResultSet.summary
```

The end part shows how we should run the queue network. We have to read permanently data from the output stream generated by the second work station. It initiates the process of receiving data from the queue located in a space between the both work stations. The corresponding queue processor begins requesting the first work station, which in its turn initiates the process of receiving data from the first queue, which processor begins reading data from the input random stream of data distributed exponentially.

The simulation experiment file mostly mimics the last return function, where the result sources are defined.

```
// File WorkStationsInSeries/RunExperiment.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.dll"

#load "Model.fsx"

open Model

open System
open System.IO

open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Web
```

```
let experiment = Experiment ()

experiment.Specs <- Model.specs
experiment.RunCount <- 1000

let queueSeries1 = ResultSet.findByName "queue1"
let queueSeries2 = ResultSet.findByName "queue2"

let serverSeries1 = ResultSet.findByName "workStation1"
let serverSeries2 = ResultSet.findByName "workStation2"

let timerSeries = ResultSet.findByName "arrivalTimer"

let providers =
    [ExperimentProvider.experimentSpecs;
     ExperimentProvider.queue queueSeries1;
     ExperimentProvider.server serverSeries1;
     ExperimentProvider.queue queueSeries2;
     ExperimentProvider.server serverSeries2;
     ExperimentProvider.arrivalTimer timerSeries]

experiment.RenderHtml (Model.model, providers)
    |> Async.RunSynchronously
```

The experiment file creates a lot of information. For brevity, only two charts are provided on figures 6.1 and 6.2. The first chart shows the trend and confidence intervals for the first queue size, while the second chart shows the lost item count for the first queue.

6.9 Example: A Machine Tool with Breakdowns

The next example[5] illustrates the use of resource preemption.

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

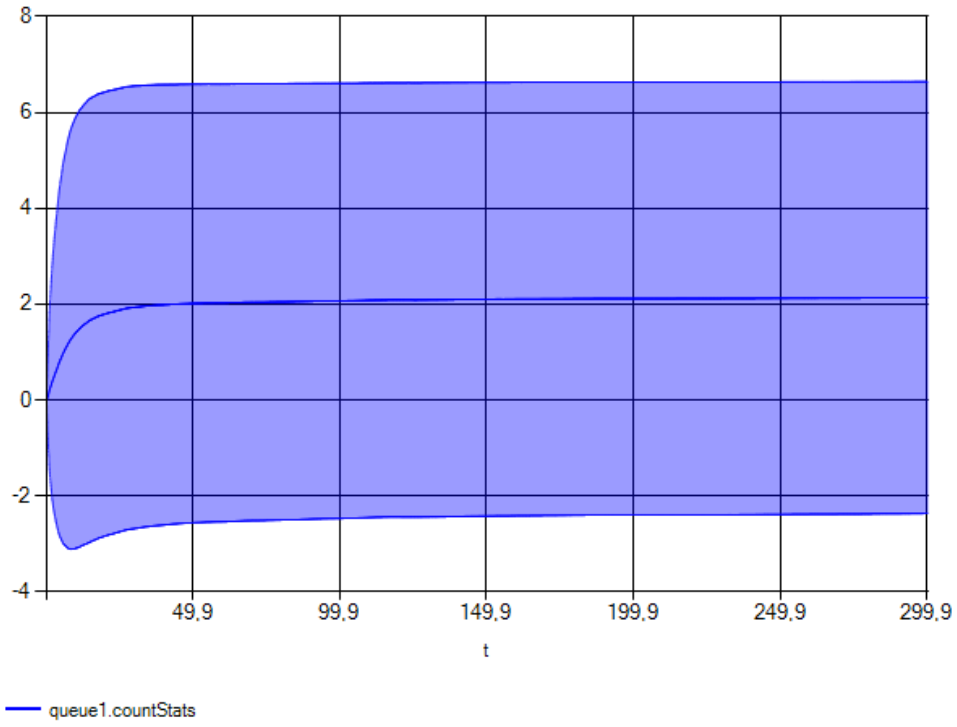


Figure 6.1: The first queue size trend and confidence intervals.

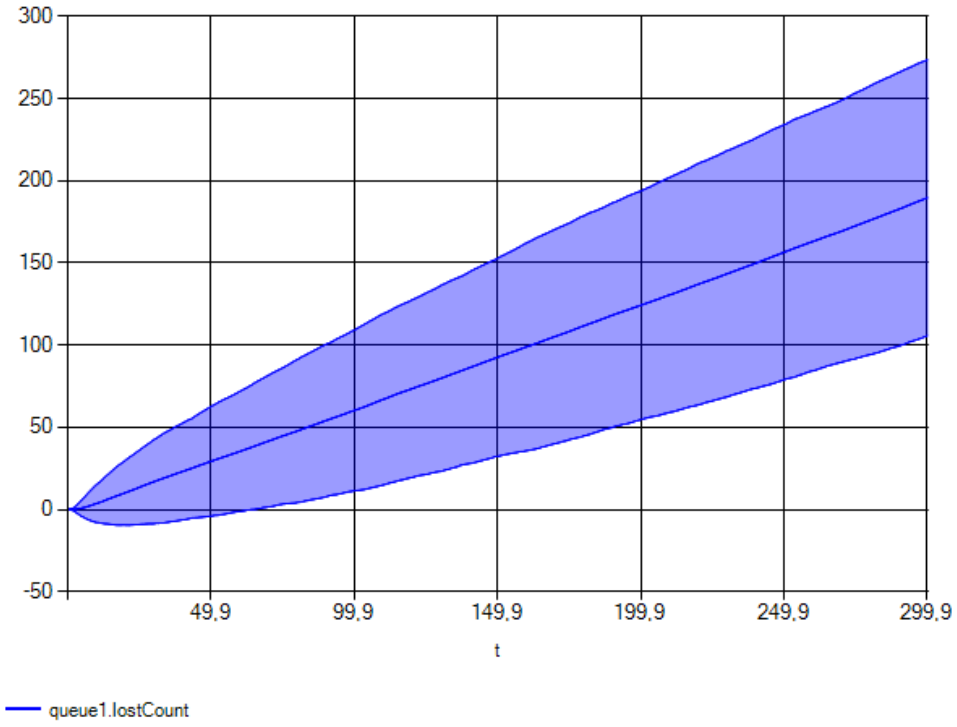


Figure 6.2: The lost item count for the first queue.

of the shop and are considered to have left the machine tool area.

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

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

We create two difference processes. The first process models the processing of jobs, while another models breakdowns. The both processes try to acquire a shared resource but with different preemption priorities, where the breakdown and the further repairing of the machine tool have a higher priority than the processing of jobs.

```
// File MachineBreakdowns/Model.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

open Simulation.Aivika
open Simulation.Aivika.Queues
open Simulation.Aivika.Results

/// the simulation specs
let specs = {

    StartTime=0.0; StopTime=500.0; DT=0.1;
```

```
    Method=RungeKutta4; GeneratorType=StrongGenerator
}

/// How often do jobs arrive to a machine tool (exponential)?
let jobArrivingMu = 1.0

/// A mean of time to process a job (normal).
let jobProcessingMu = 0.5

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

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

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

/// A mean of time between breakdowns (normal).
let breakdownMu = 20.0

/// The standard deviation of time between breakdowns (normal).
let breakdownSigma = 2.0

/// The mean of each of the three repair phases (Erlang).
let repairMu = 3.0 / 4.0

/// The priority of the job (less is higher)
let jobPriority = 1.0

/// A priority of the breakdown (less is higher)
let breakdownPriority = 0.0

/// The simulation model.
let model: Simulation<ResultSet> = simulation {
    // create an input queue
    let! inputQueue =
        InfiniteQueue.createUsingFCFS
            |> Eventive.runInStartTime

    // the counter of jobs completed
    let! jobsCompleted = ArrivalTimer.create

    // the counter of interrupted jobs
    let jobsInterrupted = ref 0
```

```
// create the input stream
let inputStream = Stream.randomExponential jobArrivingMu

// create a preemptible resource
let! tool = PreemptibleResource.create 1

// the machine setting up
let! machineSettingUp =
    Server.createRandomUniformPreemptible
        true minSetUpTime maxSetUpTime

// the machine processing
let! machineProcessing =
    Server.createRandomNormalPreemptible
        true jobProcessingMu jobProcessingSigma

// the machine breakdown
let machineBreakdown = proc {
    while true do
        do! Proc.randomNormal_ breakdownMu breakdownSigma
        use! h =
            PreemptibleResource.takeWithPriority
                breakdownPriority tool
        do! Proc.randomErlang_ repairMu 3
}

// start the process of breakdowns
do! machineBreakdown
        |> Proc.runInStartTime

// update a counter of job interruptions
do! machineProcessing
        |> Server.taskPreempting
        |> Signal.add (fun a -> eventive { incr jobsInterrupted })
        |> Eventive.runInStartTime

// define the queue network
let network =
    InfiniteQueue.processor inputQueue >>
    Processor.within
        (PreemptibleResource.requestWithPriority jobPriority
                                                 tool) >>
    Server.processor machineSettingUp >>
    Server.processor machineProcessing >>
```

```
        Processor.within
            (PreemptibleResource.release tool) >>
        ArrivalTimer.processor jobsCompleted

    // start the machine tool
    do! network inputStream
            |> Stream.sink
            |> Proc.runInStartTime

    // return the simulation results in start time
    return
        [ResultSource.From ("inputQueue",
            inputQueue, "the queue of jobs");
         ResultSource.From ("machineSettingUp",
            machineSettingUp, "the machine setting up");
         ResultSource.From ("machineProcessing",
            machineProcessing, "the machine processing");
         ResultSource.From ("jobsInterrupted",
            jobsInterrupted, "a counter of the interrupted jobs");
         ResultSource.From ("jobsCompleted",
            jobsCompleted, "a counter of the completed jobs")]
                |> ResultSet.create
}

let modelSummary =
    model |> Simulation.map ResultSet.summary
```

The simulation experiment file mainly repeats the names of sources returned from the model. The Monte-Carlo simulation contains 1000 runs.

```
// File MachineBreakdowns/RunExperiment.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.dll"

#load "Model.fsx"

open Model

open System
open System.IO
```

```
open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Web

let experiment = Experiment ()

experiment.Specs <- Model.specs
experiment.RunCount <- 1000

let inputQueue = ResultSet.findByName "inputQueue"

let machineSettingUp = ResultSet.findByName "machineSettingUp"
let machineProcessing = ResultSet.findByName "machineProcessing"

let jobsInterrupted = ResultSet.findByName "jobsInterrupted"
let jobsCompleted = ResultSet.findByName "jobsCompleted"

let providers =
    [ExperimentProvider.experimentSpecs;
     ExperimentProvider.infiniteQueue inputQueue;
     ExperimentProvider.server machineSettingUp;
     ExperimentProvider.server machineProcessing;
     ExperimentProvider.description jobsInterrupted;
     ExperimentProvider.lastValueStats jobsInterrupted;
     ExperimentProvider.arrivalTimer jobsCompleted]

experiment.RenderHtml (Model.model, providers)
    |> Async.RunSynchronously
```

We receive a lot of results. The chart displaying the utilization of the machine tool is shown on figure 6.3, but the chart of the time required to process a job is illustrated on figure 6.4.

The processing time cannot be negative, but it has so huge deviation that the chart shows negative values according to the rule of 3-sigma.

Actually, here we could save the results in CSV files with help of `TableProvider` and then analyze them in R, for example, but now we use the IronAivika capabilities that automate the process of quick analysis.

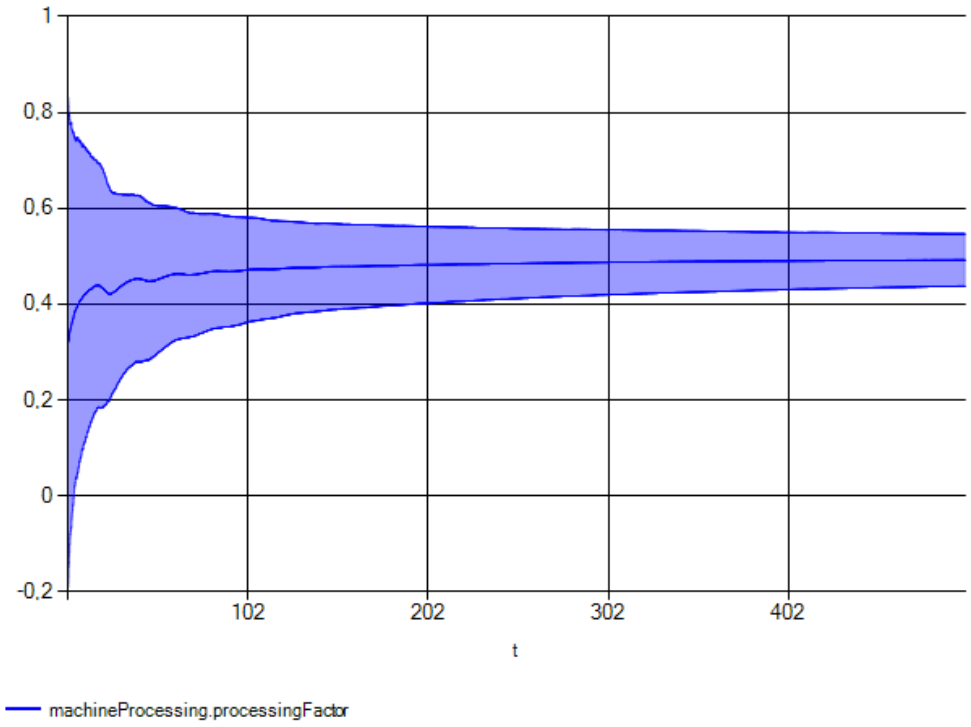


Figure 6.3: The utilization of the machine tool.

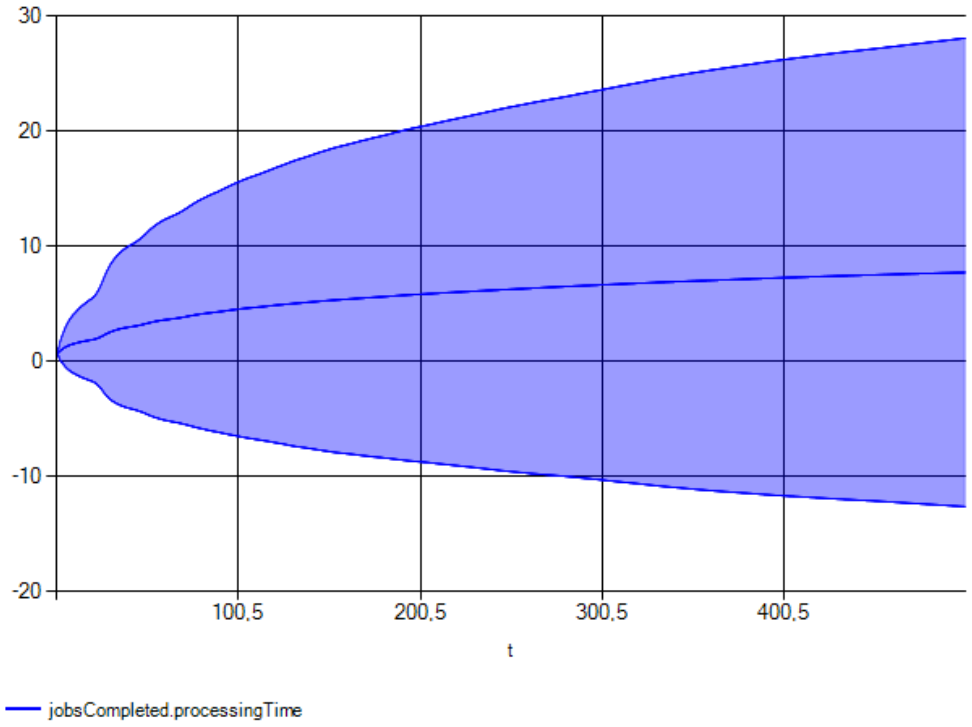


Figure 6.4: The time required to process the job.

6.10 Example: Inspection and Adjustment Stations

This example[5] illustrates how we can model a parallel work of servers. Also it shows how we can create queue networks with loopbacks.

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

The simulation model is as follows.

```
// File InspectionAdjustmentStations/Model.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

open Simulation.Aivika
```

```
open Simulation.Aivika.Queues
open Simulation.Aivika.Results

/// the simulation specs
let specs = {

    StartTime=0.0; StopTime=480.0; DT=0.1;
    Method=RungeKutta4; GeneratorType=StrongGenerator
}

/// the minimum delay of arriving the next TV set
let minArrivalDelay = 3.5

/// the maximum delay of arriving the next TV set
let maxArrivalDelay = 7.5

/// the minimum time to inspect the TV set
let minInspectionTime = 6.0

/// the maximum time to inspect the TV set
let maxInspectionTime = 12.0

/// the probability of passing the inspection phase
let inspectionPassingProb = 0.85

/// how many are inspection stations?
let inspectionStationCount = 2

/// the minimum time to adjust an improper TV set
let minAdjustmentTime = 20.0

/// the maximum time to adjust an improper TV set
let maxAdjustmentTime = 40.0

/// how many are adjustment stations?
let adjustmentStationCount = 1

let model: Simulation<ResultSet> = simulation {

    // to count the arrived TV sets for inspecting and adjusting
    let! inputArrivalTimer = ArrivalTimer.create

    // it will gather the statistics of the processing time
    let! outputArrivalTimer = ArrivalTimer.create
```

```
// define a stream of input events
let inputStream =
    Stream.randomUniform minArrivalDelay maxArrivalDelay

// create a queue before the inspection stations
let! inspectionQueue =
    InfiniteQueue.createUsingFCFS
        |> Eventive.runInStartTime

// create a queue before the adjustment stations
let! adjustmentQueue =
    InfiniteQueue.createUsingFCFS
        |> Eventive.runInStartTime

// create the inspection stations (servers)
let! inspectionStations =
    [ for i = 1 to inspectionStationCount do
        yield Server.createRandomUniform
            minInspectionTime maxInspectionTime ]
        |> Simulation.ofList

// create the adjustment stations (servers)
let! adjustmentStations =
    [ for i = 1 to adjustmentStationCount do
        yield Server.createRandomUniform
            minAdjustmentTime maxAdjustmentTime ]
        |> Simulation.ofList

// the line of parallel inspection stations
let inspectionProcessor =
    inspectionStations
        |> List.map Server.processor
        |> Processor.par

// the line of adjustment stations
let adjustmentProcessor =
    adjustmentStations
        |> List.map Server.processor
        |> Processor.par

// an output stream that comes after the inspection stations
let rec outputStream = stream {
        let xs: Stream<_> =
            inspectionQueue
                |> InfiniteQueue.dequeue
```

```
                |> Stream.repeat
                |> inspectionProcessor
        for a in xs do
            let! passed =
                Parameter.randomTrue inspectionPassingProb
                    |> Parameter.lift
            if passed then
                yield a
            else
                do! adjustmentQueue
                        |> InfiniteQueue.enqueue a
                        |> Eventive.lift
    }

// the terminal processor
and terminalProcessor =
    outputStream
        |> ArrivalTimer.processor outputArrivalTimer

// the process of adjusting TV sets
and adjustmentProcess = proc {
        let xs: Stream<_> =
            adjustmentQueue
                |> InfiniteQueue.dequeue
                |> Stream.repeat
                |> adjustmentProcessor
        for a in xs do
            do! inspectionQueue
                    |> InfiniteQueue.enqueue a
                    |> Eventive.lift
    }

// the input process
and inputProcess = proc {
        let xs: Stream<_> =
            inputStream
                |> ArrivalTimer.processor inputArrivalTimer
        for a in xs do
            do! inspectionQueue
                    |> InfiniteQueue.enqueue a
                    |> Eventive.lift
    }

// run the process of adjustment
do! adjustmentProcess
```

```
            |> Proc.runInStartTime

    // run the input process
    do! inputProcess
            |> Proc.runInStartTime

    // run the terminal processor
    do! terminalProcessor
            |> Stream.sink
            |> Proc.runInStartTime

    // return the simulation results
    return
        [ResultSource.From ("inspectionQueue", inspectionQueue,
            "the inspection queue");
         ResultSource.From ("adjustmentQueue", adjustmentQueue,
            "the adjustment queue");
         ResultSource.From ("inputArrivalTimer", inputArrivalTimer,
            "the input arrival timer");
         ResultSource.From ("outputArrivalTimer", outputArrivalTimer,
            "the output arrival timer");
         ResultSource.From ("inspectionStations", inspectionStations,
            "the inspection stations");
         ResultSource.From ("adjustmentStations", adjustmentStations,
            "the adjustment stations")]
                |> ResultSet.create
}

let modelSummary: Simulation<ResultSet> =
    model |> Simulation.map ResultSet.summary
```

We will use the following simulation experiment.

```
// File InspectionAdjustmentStations/RunExperiment.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.dll"

#load "Model.fsx"

open Model
```

```
open System
open System.IO

open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Web

let experiment = Experiment ()

experiment.Specs <- Model.specs
experiment.RunCount <- 1000

let inspectionQueue = ResultSet.findByName "inspectionQueue"
let adjustmentQueue = ResultSet.findByName "adjustmentQueue"

let inspectionStations = ResultSet.findByName "inspectionStations"
let adjustmentStations  = ResultSet.findByName "adjustmentStations"

let outputTimer = ResultSet.findByName "outputArrivalTimer"

let providers =
    [ExperimentProvider.experimentSpecs;
     ExperimentProvider.infiniteQueue inspectionQueue;
     ExperimentProvider.infiniteQueue adjustmentQueue;
     ExperimentProvider.server inspectionStations;
     ExperimentProvider.server adjustmentStations;
     ExperimentProvider.arrivalTimer outputTimer]

experiment.RenderHtml (Model.model, providers)
    |> Async.RunSynchronously
```

Some of the results are shown on figures 6.5, 6.6 and 6.7.

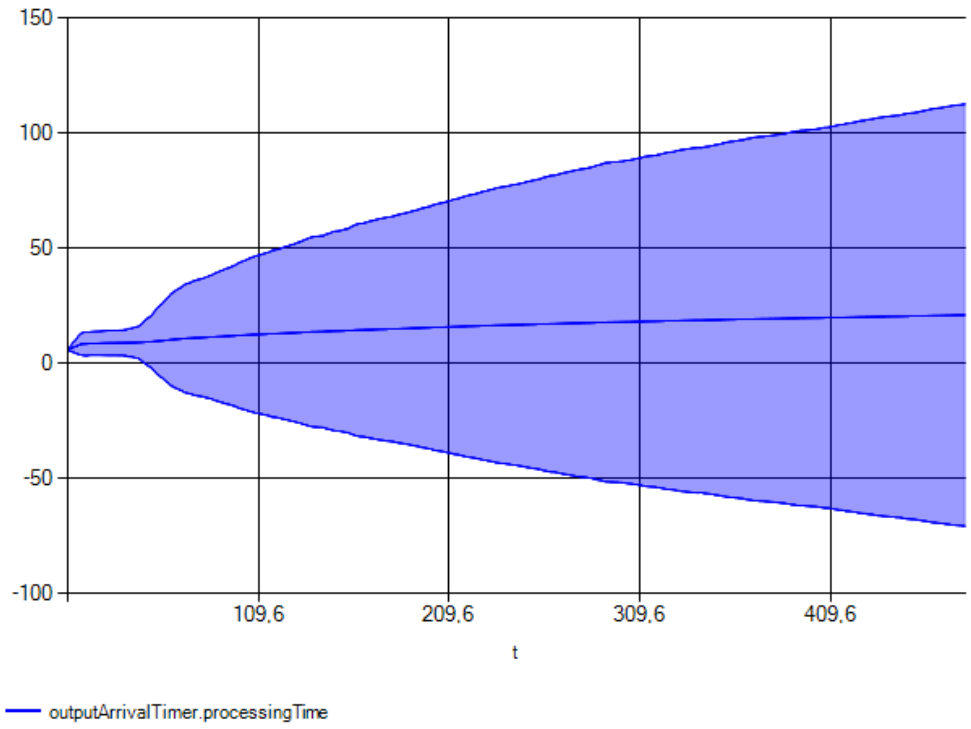


Figure 6.5: The processing time of television sets.

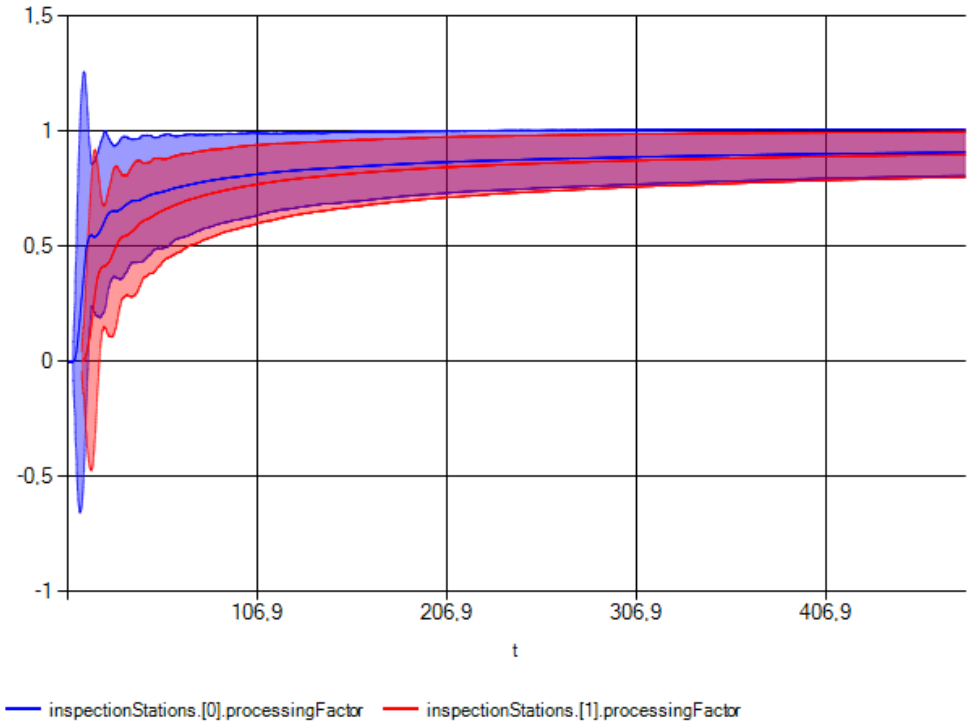


Figure 6.6: The utilization of the inspectors.

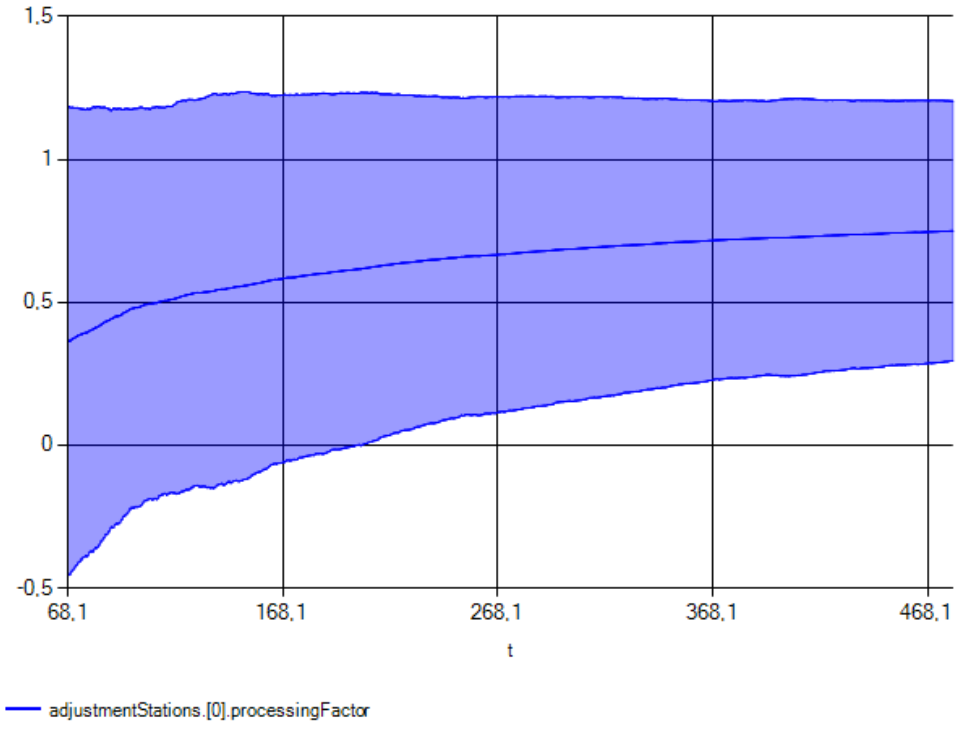


Figure 6.7: The utilization of the adjustor.

Chapter 7

System Dynamics

The IronAivika library was mainly designed and created for the field of Discrete Event Simulation. However, the library can be used for solving tasks of System Dynamics too. Moreover, the both fields can be naturally combined. Actually, the Discrete Event Simulation computations are implemented on top of the `Dynamics` computation, which is used for System Dynamics.

7.1 Memoizing Sequential Computations

The key feature that distinguishes the `Dynamics` computation from the `Eventive` one is that the modeling time flows in an unpredictable order within the former computation.

For example, the past value can be requested from the future: the initial value of the integral can be requested at any time and so on.

IronAivika solves this task by ordering and memoizing the computations in integration time points.

```
module Dynamics =

    val memo : comp:Dynamics<'a> -> Dynamics<'a>
    val memo0 : comp:Dynamics<'a> -> Dynamics<'a>
```

The both functions return a new computation that sequentially calls the input computation in the integration time points by demand and saves the values in array so that the next call in the same integration time point will

return the already calculated value without calling the input computation twice.

In other non-integration time points the values are interpolated so that the closest past integration time point is selected, which allows using the resulting computation in the discrete event simulation. Also we cannot request for the future value from the past. Otherwise, we might receive a deadlock. This is a step-wise linear interpolation.

The functions differ in that how they behave when specifying the Runge-Kutta integration method.

The `memo` function is destined for integrating the equations by the Runge-Kutta method. It consumes more memory to allocate an array that stores the values in the intermediate integration time points that are used by this integration technique.

The `memo0` function is more fast. It uses only the basic integration time points, as Euler's integration method would be used.

Usually, the `memo` function is needed only for integrals, while the `memo0` is suitable for other cases.

To emphasize the difference between the integration time points, there are two functions that make the idea more clear.

```
module Dynamics =

    val interpolate : comp:Dynamics<'a> -> Dynamics<'a>
    val discrete : comp:Dynamics<'a> -> Dynamics<'a>
```

The `interpolate` function is used by the `memo` function. This is such an interpolation, when the values in the integration time points including the intermediate integration time points are returned as they are, while in other time points we return a value for the closest past integration time point.

The `discrete` function is used by the `memo0` function. This is the interpolation, where only the basic integration time points are used, i.e. those time points that are defined by Euler's method.

Some simulation software tools for System Dynamics such as Vensim define functions that have an effect, when the value returned by the function changes only in the integration time point regardless on the integration method used. The `discrete` function gives namely this effect.

To complete the picture, there is a function that returns the initial value of the computation that was defined in the start time, but only now the initial value is returned in the current simulation time.

```
module Dynamics =
    val runInStartTime : comp:Dynamics<'a> -> Simulation<'a>
```

For example, this function can be used for receiving the initial value of the integral.

Returning to the memoization functions, they can be used not only for integrals. They can also be used for creating random processes that can be defined in the differential and difference equations of System Dynamics.

```
module Dynamics =

    val memoRandomUniform : minimum:Dynamics<float>
                            -> maximum:Dynamics<float>
                            -> Dynamics<float>

    val memoRandomUniformInt : minimum:Dynamics<int>
                               -> maximum:Dynamics<int>
                               -> Dynamics<int>

    val memoRandomNormal : mean:Dynamics<float>
                           -> deviation:Dynamics<float>
                           -> Dynamics<float>

    val memoRandomExponential : mean:Dynamics<float> -> Dynamics<float>

    val memoRandomErlang : beta:Dynamics<float>
                           -> m:Dynamics<int>
                           -> Dynamics<float>

    val memoRandomPoisson : mean:Dynamics<float> -> Dynamics<int>

    val memoRandomBinomial : prob:Dynamics<float>
                             -> trials:Dynamics<int>
                             -> Dynamics<int>
```

They are based on the `memo0` function, i.e. the result changes only in the integration time point.

7.2 Table Function

Many simulation models of System Dynamics use graphical functions based on tables of pairs (x,y).

```
[<Sealed>]
type Table =

    new : xys:(float * float) [] -> Table

    member Lookup : x:float -> float
    member LookupStepwise : x:float -> float
```

The first lookup method uses the linear interpolation, while the second method uses a step-wise linear interpolation.

To make the table functions more easy-to-use, the library defines convenient helper functions to be used in the differential and difference equations.

```
module Dynamics =

    val lookup : x:Dynamics<float> -> tbl:Table -> Dynamics<float>
    val lookupStepwise : x:Dynamics<float> -> tbl:Table -> Dynamics<float>
```

For example, the first function could be trivially defined as

```
module Dynamics =

    let lookup (x: Dynamics<float>) (t: Table)  =
        dynamics {
            let! a = x
            return t.Lookup (a)
        }
```

Note the use of the computation expression syntax. It literally means that you can include your own functions in the differential and difference equations, for they are actually a system of `Dynamics` computations.

7.3 Differential Equations

The integral function signature was stated before and it is repeated here again for convenience.

```
module SD =

    val integ : derivative:Lazy<Dynamics<float>>
                -> init:Dynamics<float>
                -> Dynamics<float>
```

It creates an integral by the specified derivative and initial value.

Comparing to specialized simulation software tools, this function is rather slow, but it works[1]. Moreover, it can be used in the combined discrete continuous simulation models.

We can create the ordinary differential equations of almost any complexity based on the `integ` function.

Below is provided the implementation of the n'th order exponential smooth function.

```
module SD =

    let smoothN (x: Dynamics<float>) (t: Lazy<Dynamics<float>>) n =

        let rec s = [|
            for k = 0 to n-1 do
                if k = 0 then
                    yield integ (lazy ((x - s.[k]) /
                                       (t.Value / (float n)))) x
                else
                    yield integ (lazy ((s.[k-1] - s.[k]) /
                                       (t.Value / (float n)))) x |]
        in s.[n-1]
```

The key point is that the integrals are just `Dynamics` computations that can be combined in a very sophisticated manner.

Earlier we saw a few examples of the discrete event simulation models that extensively used the computation expression syntax of F#. The same syntax can be used for extending the differential equations, which allows embed your own functions in the equations.

For example, see a possible implementation of the table lookup function in section 7.2.

Regarding the combination with discrete event simulation models, there are two points, at least. An arbitrary `Dynamics` computation can be transformed to the `Eventive` computation. Then, we can use the `Var` data type to create entities that could be used in the differential equations, but which would be updated from the discrete event simulation.

[1]The Equation Compiler version of VisualAivika uses another approach for integrating the differential equations, which is highly optimized for the speed of execution.

7.4 Difference Equations

The difference equations are much like the differential ones, only another function is used for creating a sum by the specified difference and initial value.

```
module SD =

    val diffsum: difference:Lazy<Dynamics<float>>
                 -> init:Dynamics<float>
                 -> Dynamics<float>
```

7.5 Example: Parametric Financial Model

The next example illustrates a parametric model in combination with the Monte-Carlo simulation. The results received can be useful for the Sensitivity Analysis. The approach described works for the discrete event simulation too.

We will take the financial model[12] described in Vensim 5 Modeling Guide, Chapter Financial Modeling and Risk. Probably, the best way to describe the model is just to show its equations.

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

```
module SD =

    let npv stream rate init factor =
        let rec dt' = Parameter.dt |> Parameter.lift
        and df = integ (lazy (- df * rate)) (num 1.0)
        and accum = integ (lazy (stream * df)) init
        in (accum + dt' * stream * df) * factor
```

Also we need a helper conditional combinator that allows simplifying the equations in some cases.

```
module SD =

    val ifThenElse: cond:Dynamics<bool>
                    -> thenPart:Dynamics<'a>
                    -> elsePart:Dynamics<'a>
                    -> Dynamics<'a>
```

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

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

```
// File Financial/Model.fsx

#nowarn "40"

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

open Simulation.Aivika
open Simulation.Aivika.SD
open Simulation.Aivika.Results

/// The simulation specs
let specs =
    { StartTime = 0.0;
      StopTime = 5.0;
      DT = 0.015625;
      Method = RungeKutta4;
      GeneratorType = StrongGenerator }

/// The model parameters.
type Parameters =
    { TaxDepreciationTime    : Parameter<float>;
      TaxRate                : Parameter<float>;
      AveragePayableDelay    : Parameter<float>;
      BillingProcessingTime  : Parameter<float>;
      BuildingTime           : Parameter<float>;
      DebtFinancingFraction  : Parameter<float>;
      DebtRetirementTime     : Parameter<float>;
      DiscountRate           : Parameter<float>;
      FractionalLossRate     : Parameter<float>;
      InterestRate           : Parameter<float>;
      Price                  : Parameter<float>;
      ProductionCapacity     : Parameter<float>;
      RequiredInvestment     : Parameter<float>;
      VariableProductionCost : Parameter<float> }

/// The default model parameters.
```

```
let defaultParams =
    { TaxDepreciationTime    = parameter.Return 10.0;
      TaxRate                = parameter.Return 0.4;
      AveragePayableDelay    = parameter.Return 0.09;
      BillingProcessingTime  = parameter.Return 0.04;
      BuildingTime           = parameter.Return 1.0;
      DebtFinancingFraction  = parameter.Return 0.6;
      DebtRetirementTime     = parameter.Return 3.0;
      DiscountRate           = parameter.Return 0.12;
      FractionalLossRate     = parameter.Return 0.06;
      InterestRate           = parameter.Return 0.12;
      Price                  = parameter.Return 1.0;
      ProductionCapacity     = parameter.Return 2400.0;
      RequiredInvestment     = parameter.Return 2000.0;
      VariableProductionCost = parameter.Return 0.6 }

/// Random parameters for the Monte-Carlo simulation.
let randomParams =
    let averagePayableDelay    = Parameter.randomUniform 0.07 0.11
    let billingProcessingTime  = Parameter.randomUniform 0.03 0.05
    let buildingTime           = Parameter.randomUniform 0.8 1.2
    let fractionalLossRate     = Parameter.randomUniform 0.05 0.08
    let interestRate           = Parameter.randomUniform 0.09 0.15
    let price                  = Parameter.randomUniform 0.9 1.2
    let productionCapacity     = Parameter.randomUniform 2200.0 2600.0
    let requiredInvestment     = Parameter.randomUniform 1800.0 2200.0
    let variableProductionCost = Parameter.randomUniform 0.5 0.7
    { defaultParams with
        AveragePayableDelay    = Parameter.memo averagePayableDelay;
        BillingProcessingTime  = Parameter.memo billingProcessingTime;
        BuildingTime           = Parameter.memo buildingTime;
        FractionalLossRate     = Parameter.memo fractionalLossRate;
        InterestRate           = Parameter.memo interestRate;
        Price                  = Parameter.memo price;
        ProductionCapacity     = Parameter.memo productionCapacity;
        RequiredInvestment     = Parameter.memo requiredInvestment;
        VariableProductionCost = Parameter.memo variableProductionCost }

/// This is the model itself that returns experimental data.
let model (ps: Parameters) : Simulation<ResultSet> = simulation {

    let get (x: Parameter<_>) : Dynamics<_> = Parameter.lift x

    let taxDepreciationTime    = get ps.TaxDepreciationTime
    let taxRate                = get ps.TaxRate
```

```
let averagePayableDelay    = get ps.AveragePayableDelay
let billingProcessingTime  = get ps.BillingProcessingTime
let buildingTime           = get ps.BuildingTime;
let debtFinancingFraction  = get ps.DebtFinancingFraction
let debtRetirementTime     = get ps.DebtRetirementTime
let discountRate           = get ps.DiscountRate
let fractionalLossRate     = get ps.FractionalLossRate
let interestRate           = get ps.InterestRate
let price                  = get ps.Price
let productionCapacity     = get ps.ProductionCapacity
let requiredInvestment     = get ps.RequiredInvestment
let variableProductionCost = get ps.VariableProductionCost

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

let rec bookValue =
    integ (lazy (newInvestment - taxDepreciation)) (num 0.0)
and taxDepreciation = bookValue / taxDepreciationTime
and taxableIncome =
    grossIncome - directCosts - losses
        - interestPayments - taxDepreciation
and production = availableCapacity
and availableCapacity =
    ifThenElse (Dynamics.time .>=. buildingTime)
        productionCapacity (num 0.0)
and accountsReceivable =
    integ (lazy (billings - cashReceipts - losses))
        (billings / (num 1.0 / averagePayableDelay
            + fractionalLossRate))
and awaitingBilling =
    integ (lazy (price * production - billings))
        (price * production * billingProcessingTime)
and billings = awaitingBilling / billingProcessingTime
and borrowing = newInvestment * debtFinancingFraction
and cashReceipts = accountsReceivable / averagePayableDelay
and debt =
    integ (lazy (borrowing - principalRepayment)) (num 0.0)
and directCosts = production * variableProductionCost
and grossIncome = billings
and interestPayments = debt * interestRate
and losses = accountsReceivable * fractionalLossRate
and netCashFlow =
    cashReceipts + borrowing - newInvestment
        - directCosts - interestPayments
        - principalRepayment - taxes
```

```
    and netIncome = taxableIncome - taxes
    and newInvestment =
        ifThenElse (Dynamics.time .>=. buildingTime)
            (num 0.0) (requiredInvestment / buildingTime)
    and npvCashFlow =
        npv netCashFlow discountRate (num 0.0) (num 1.0)
    and npvIncome =
        npv netIncome discountRate (num 0.0) (num 1.0)
    and principalRepayment = debt / debtRetirementTime
    and taxes = taxableIncome * taxRate

    return
        [ResultSource.From ("netIncome",
            netIncome, "Net income");
         ResultSource.From ("netCashFlow",
            netCashFlow, "Net cash flow");
         ResultSource.From ("npvIncome",
            npvIncome, "NPV income");
         ResultSource.From ("npvCashFlow",
            npvCashFlow, "NPV cash flow")]
                |> ResultSet.create
}
```

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

The point is that not only ODEs can be parametric! There is no any difference, whether we integrate numerically, or run the discrete event simulation, or simulate the agents. The external parameters are just `Parameter` computations that can be used within other simulation computations.

Returning to this example, we will use the following simulation experiment.

```
// File Financial/RunExperiment.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.dll"

#load "Model.fsx"
```

```
open Model

open System
open System.IO

open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Web

let specs = Model.specs
let model = Model.model Model.randomParams

let experiment = Experiment ()

experiment.Specs <- specs
experiment.RunCount <- 1000

let income =
    [ResultSet.findByName "netIncome";
     ResultSet.findByName "netCashFlow"]
        |> ResultTransform.concat

let cashFlow =
    [ResultSet.findByName "npvIncome";
     ResultSet.findByName "npvCashFlow"]
        |> ResultTransform.concat

let providers =
    [ExperimentProvider.experimentSpecs;
     ExperimentProvider.description income;
     ExperimentProvider.deviationChart income
     ExperimentProvider.lastValueStats income;
     ExperimentProvider.lastValueHistogram income;
     ExperimentProvider.description cashFlow;
     ExperimentProvider.deviationChart cashFlow;
     ExperimentProvider.lastValueStats cashFlow;
     ExperimentProvider.lastValueHistogram cashFlow]

experiment.RenderHtml (model, providers)
    |> Async.RunSynchronously
```

The resulting deviation charts are shown on figures 7.1 and 7.2.

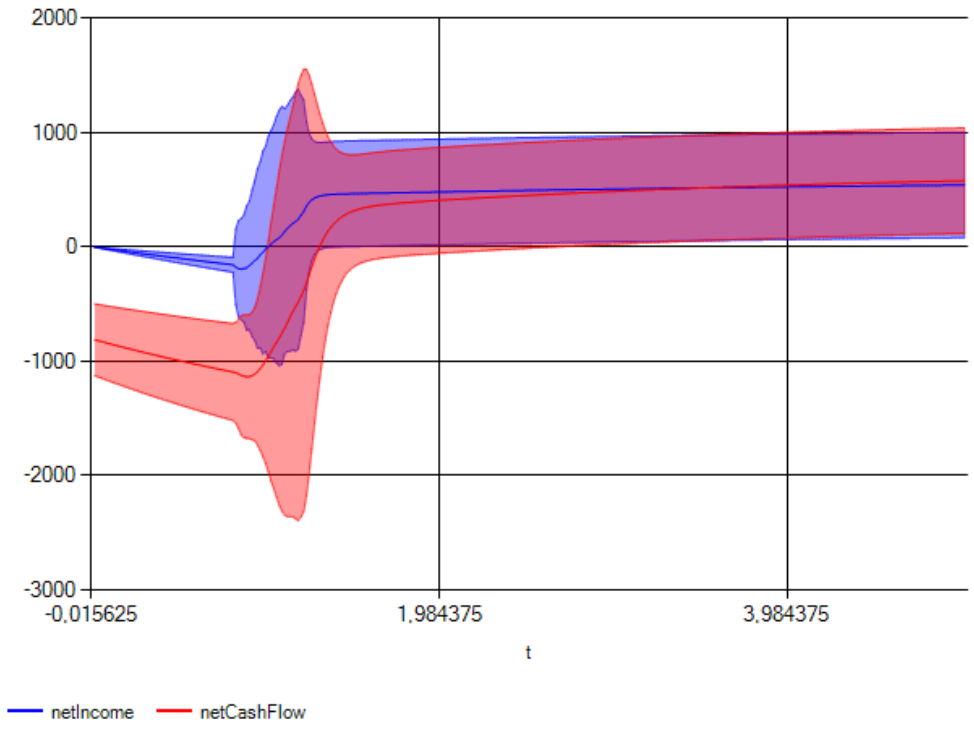


Figure 7.1: The net income and net cash flow.

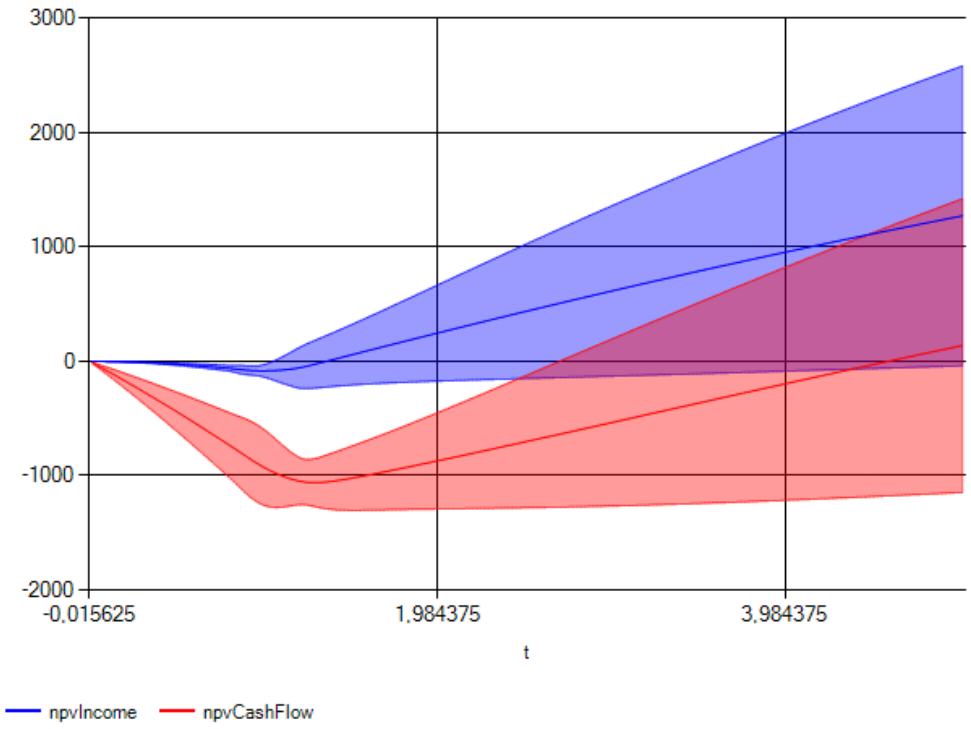


Figure 7.2: The NPV income and cash flow.

7.6 Example: Linear Array

This example illustrates the use of arrays. There is no need in special support for them. They can be naturally used together with the simulation computations.

Let us take model Linear Array from Berkeley Madonna[3] to demonstrate the main idea.

```
// File LinearArray/Model.fsx

#nowarn "40"

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

open Simulation.Aivika
open Simulation.Aivika.SD
open Simulation.Aivika.Results

let specs =
    { StartTime = 0.0;
      StopTime = 500.0;
      DT = 0.1;
      Method = RungeKutta4;
      GeneratorType = StrongGenerator }

let model (n: int) : Simulation<ResultSet> = simulation {

    let rec m : Dynamics<float> array =
        [| for i = 1 to n do
            yield integ
                (lazy (q
                    + k * (c.[i - 1] - c.[i])
                    + k * (c.[i + 1] - c.[i])))
                    (num 0.0) |]
    and c : Dynamics<float> array =
        [| for i = 0 to n + 1 do
            if i = 0 || i = n + 1 then
                yield (num 0.0)
            else
                yield (m.[i - 1] / v) |]
    and q = 1.0
```

```
    and k = 2.0
    and v = 0.75

    return
        [ResultSource.From("M", m, "M");
         ResultSource.From("C", c, "C")]
            |> ResultSet.create
}
```

Here we create two linear arrays `M` and `C`, where the first array consists of integrals. Similarly, we could use arrays in the discrete event simulation or agent-based model.

The simulation experiment shows the both arrays.

```
// File LinearArray/RunExperiment.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.dll"

#load "Model.fsx"

open Model

open System
open System.IO

open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Web

let experiment = Experiment ()

experiment.Specs <- Model.specs
experiment.RunCount <- 1

let m  = ResultSet.findByName "M"
let c  = ResultSet.findByName "C"

let provider1 = TimeSeriesProvider ()
```

```
let provider2 = TimeSeriesProvider ()

provider1.Series <- m
provider2.Series <- c

let providers =
    [ExperimentProvider.experimentSpecs;
     provider1 :> IExperimentProvider<TextWriter>;
     provider2 :> IExperimentProvider<TextWriter>]

experiment.RenderHtml (Model.model 51, providers)
    |> Async.RunSynchronously
```

The charts are provided on figures 7.3 and 7.4.

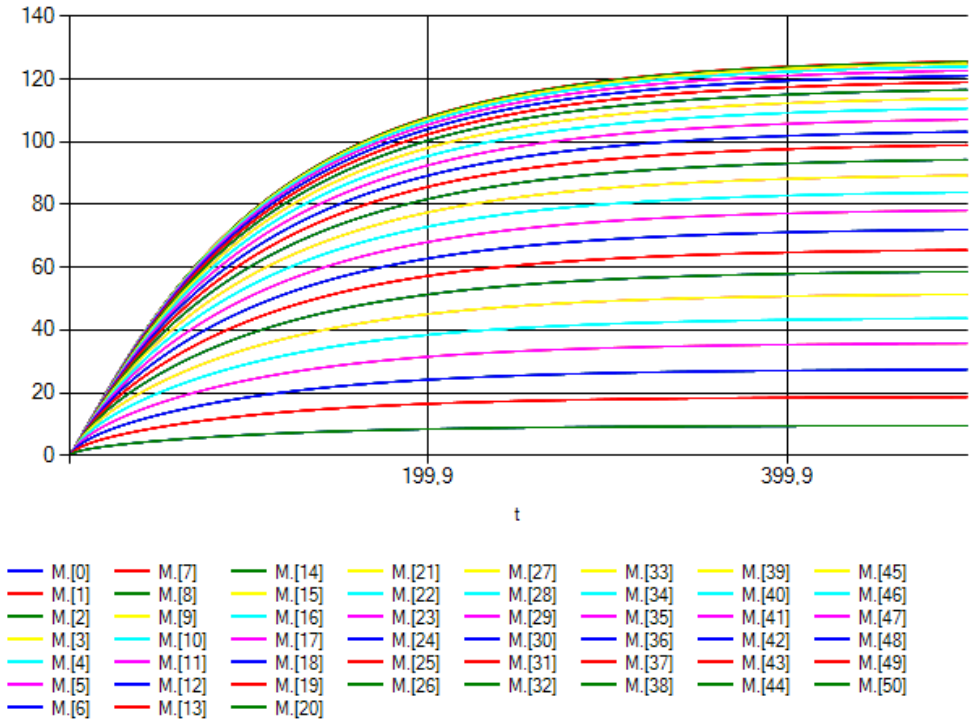


Figure 7.3: The array of integrals.

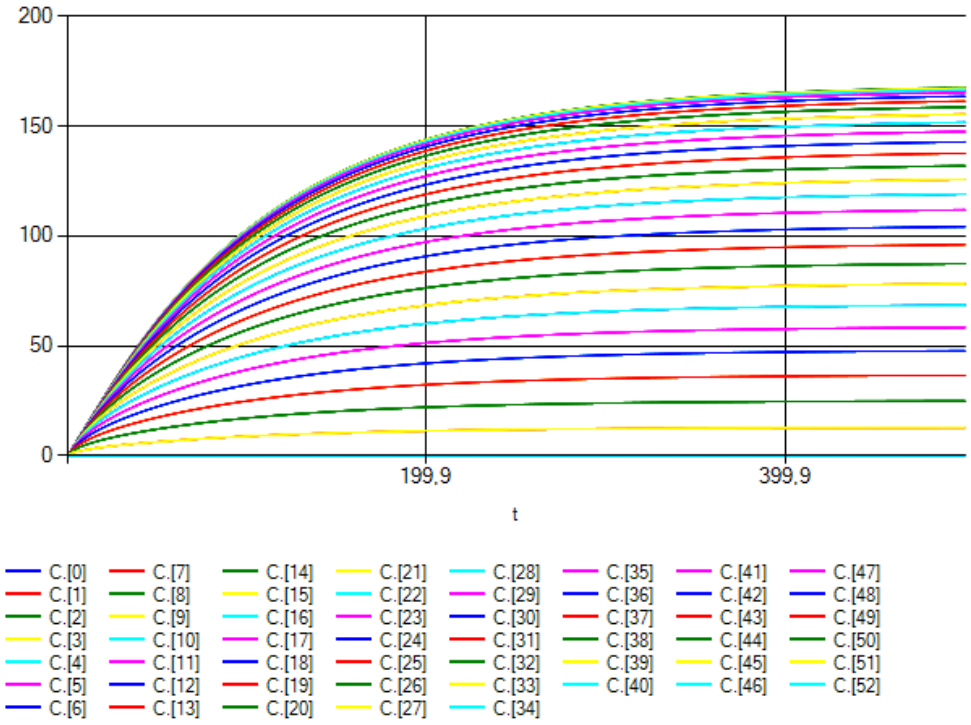


Figure 7.4: The second linear array.

Chapter 8

Agent-based Modeling

IronAivika supports the Agent-based Modeling[10] on basic level, and this support is well integrated with other simulation computations of the library.

8.1 Agents and States

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

We create new agents within the `Simulation` computation, but define the states as dependent objects.

```
type [<Sealed>] Agent =
    ...

and [<AbstractClass>] AgentState =

    new : agent:Agent -> AgentState
    new : parent:AgentState -> AgentState
    ...

module Agent =
    val create : Simulation<Agent>
```

Only one of the states can be selected for each agent at the current modeling time. All ancestor states remain active if they were active before,

or they become active if they were deactivated. Other states are deactivated if they were active, on the contrary.

```
module Agent =

    val selectedState : agent:Agent -> Eventive<AgentState option>
    val selectedStateChanged : agent:Agent -> Signal<AgentState option>
    val selectedStateChanged_ : agent:Agent -> Signal<unit>

module AgentState =
    val select : state:AgentState -> Eventive<unit>
```

The `selectedState` function returns the currently selected state or `None` if the agent was not yet initiated, but the `select` function allows us to select a new state. The both functions return actions within the `Eventive` computation, which means that the state selection is always synchronized with the event queue.

We can assign the `Eventive` handlers to be performed when activating or deactivating the specified third state during such a selection. They are defined as the object methods.

```
type [<AbstractClass>] AgentState =

    abstract Activate : unit -> Eventive<unit>
    abstract Deactivate : unit -> Eventive<unit>

    default Activate: unit -> Eventive<unit>
    default Deactivate: unit -> Eventive<unit>

    ...
```

When the target state is selected, we can define the next target state if needed.

```
type [<AbstractClass>] AgentState =

    abstract Transit : unit -> Eventive<AgentState option>
    default Transit: unit -> Eventive<AgentState option>

    ...
```

What differs the agents from other simulation concepts is an ability to assign so called *timeout* and *timer handlers*. The timeout handler is an

`Eventive` computation which is actuated in the specified time interval if the sate remains active. The timer handler is similar, but only the handler is repeated while the state still remains active. Therefore, the timeout handler accepts the time as a pure value, while the timer handler recalculates the time interval within the `Eventive` computation after each successful actualization.

```
module AgentState =

    val addTimeout : Time
                     -> Eventive<unit>
                     -> AgentState
                     -> Eventive<unit>

    val addTimer : Eventive<Time>
                   -> Eventive<unit>
                   -> AgentState
                   -> Eventive<unit>
```

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

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

8.2 Example: Agent-based Modeling

To illustrate the use of agents, let us take the Bass Diffusion model from the AnyLogic documentation [10].

> The model describes a product diffusion process. Potential adopters of a product are influenced into buying the product by advertising and by word of mouth from adopters, those who have already purchased the new product. Adoption of a new product driven by word of mouth is likewise an epidemic.

> Potential adopters come into contact with adopters through social interactions. A fraction of these contacts results in the purchase of the new product. The advertising causes a constant fraction of the potential adopter population to adopt each time period.

The simulation model is as follows.

```
// File BassDiffusion/Model.fsx

#nowarn "40"

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

open System
open System.Collections.Generic

open Simulation.Aivika
open Simulation.Aivika.Results

let n = 100     // 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;
      GeneratorType = StrongGenerator }

type PersonContext =
    { PotentialAdopters: int ref;
      Adopters: int ref;
      Persons: List<Person> }

and Person (ctx: PersonContext, agent: Agent) =

    let rec potentialAdopter =
        { new AgentState (agent) with
            member x.Activate () = eventive {
                incr ctx.PotentialAdopters
```

```
            // create a timeout that will hold
            // while the state is active

            let! t = Parameter.randomExponential
                        (1.0 / advertisingEffectiveness)
                            |> Parameter.lift

            do! potentialAdopter
                    |> AgentState.addTimeout t
                        (AgentState.select adopter)
        }

        member x.Deactivate () = eventive {
            decr ctx.PotentialAdopters
        }
    }
and adopter =
    { new AgentState (agent) with
        member x.Activate () = eventive {
            incr ctx.Adopters

            // create a timer that will hold
            // while the state is active

            let t = Parameter.randomExponential
                        (1.0 / contactRate)
                            |> Parameter.lift

            let m =
                eventive {
                    let! i = Parameter.randomUniformInt
                                0 (ctx.Persons.Count - 1)
                                    |> Parameter.lift

                    do! ctx.Persons.[i].Buy ()
                }

            do! adopter |> AgentState.addTimer t m
        }

        member x.Deactivate () = eventive {
            decr ctx.Adopters
        }
    }
```

```
    member x.Agent = agent
    member x.PotentialAdopter = potentialAdopter
    member x.Adopter = adopter

    member private x.Buy () = eventive {

        let! st = Agent.selectedState agent

        if st = Some potentialAdopter then

            let! x = Parameter.randomTrue adoptionFraction
                        |> Parameter.lift

            if x then
                do! AgentState.select adopter
    }

    member x.Init () = eventive {
        ctx.Persons.Add (x)
        do! AgentState.select potentialAdopter
    }

let model: Simulation<ResultSet> = simulation {

    let ctx =
        { PotentialAdopters = ref 0;
          Adopters = ref 0;
          Persons  = List<_> () }

    for i = 1 to n do
        let! agent = Agent.create
        let person = Person (ctx, agent)
        do! person.Init () |> Eventive.runInStartTime

    return
        [ResultSource.From ("potentialAdopters",
            ctx.PotentialAdopters, "Potential Adopters");
         ResultSource.From ("adopters",
            ctx.Adopters, "Adopters")]
                |> ResultSet.create
}
```

The reader can notice that the model uses the same computations that we used for the ordinary differential equations and discrete event simula-

tion.

Now we will define an experiment by trying to plot the deviation chart for potential adopters and adopters. Unlike other cases, we will launch a hundred of simulation runs as this model requires more extensive computations because of multiple agents.

```
// File BassDiffusion/RunExperiment.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.dll"

#load "Model.fsx"

open Model

open System
open System.IO

open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Web

let experiment = Experiment ()

experiment.Specs <- Model.specs
experiment.RunCount <- 100

let series =
    [ResultSet.findByName "potentialAdopters";
     ResultSet.findByName "adopters"]
        |> ResultTransform.concat

let providers =
    [ExperimentProvider.experimentSpecs;
     ExperimentProvider.description series;
     ExperimentProvider.deviationChart series]

experiment.RenderHtml (Model.model, providers)
    |> Async.RunSynchronously
```

The resulting chart is shown on figure 8.1

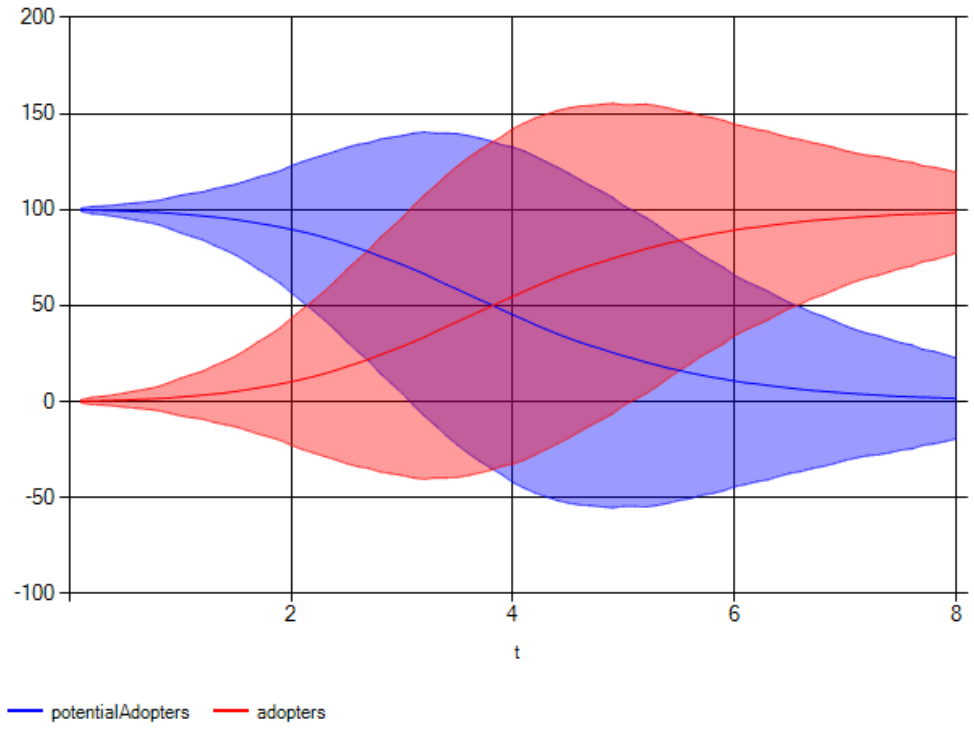


Figure 8.1: The potential adopters and adopters.

Chapter 9

Blocks

I extensively use blocks for my own implementation of Discrete Event Simulation in VisualAivika. If you want to write extension modules for your VisualAivika models then you will certainly need to read this chapter.

The represented `Block` computations were inspired by the GPSS modeling language[6]. It has turned out that the blocks are so wonderful for creating discrete event simulation models that many other simulation entities from IronAivika were considered to be obsolete and outdated. If you want to create new discrete event simulation models with help of IronAivika then it is highly recomended to use namely blocks.

9.1 Block Computation

The `Block` computation is just a function that takes the input and then returns the corresponding output within the discontinuous process.

```
type Block<'a, 'b> = Block of ('a -> Proc<'b>)
```

This fact is directly reflected in the run function that allows involving `Block` computations in simulation.

```
module Block =
    val run: input:'a -> comp:Block<'a, 'b> -> Proc<'b>
```

There are different ways how the blocks can be created, but the most fundamental ones are as follows. Here the names are related to word *arrow*.

```
module Block =

    val inline arr: f:('a -> 'b) -> Block<'a, 'b>
    val inline arrc: f:('a -> Proc<'b>) -> Block<'a, 'b>
```

These functions allow us to create arbitrary blocks from the simulation computations considered before. Therefore, the blocks are naturally involved in simulation. They are processed by the same event queue. They can be a part of the same Monte-Carlo simulation.

The key feature of the `Block` computations it that they can create a composition, where the second block follows the first one in the processing of input values.

```
module Block =

    val inline compose: comp1:Block<'a, 'b>
                        -> comp2:Block<'b, 'c>
                        -> Block<'a, 'c>

[<AutoOpen>]
module BlockComposition =

    val inline (>>>): comp1:Block<'a, 'b>
                      -> comp2:Block<'b, 'c>
                      -> Block<'a, 'c>
```

The infix `(>>>)` operator is actually a shorthand version of the mentioned `Block.compose` function. The operator is more handy for creating chains of `Block` computations, where the blocks process their input sequentially. But such a processing can be parallel for different input values!

```
let b1: Block<'a, 'a>
let b2: Block<'a, unit>
let b3: Block<'a, unit> = b1 >>> b2
```

We are mostly interested in the block computations that have such types.

Here the `b1` computation is associated with something that takes a transact (order) of type `'a`, simulates some activity and then returns the same transact, although in rare cases it could transform the transact by returning something else of type `'b`, when the type of the computation would be `Block<'a, 'b>`.

Blocks `b2` and `b3` are *terminal* here. Such a block either terminates the processing of the specified transact or creates a loopback.

```
module Block =
    val terminate<'a> : Block<'a, unit>

module Blocks =
    val inline transfer: comp:Block<'a, unit> -> Block<'a, 'b>
```

Here the `Block.terminate` function terminates the processing for every input transact. The `Blocks.transfer` allows us to create the loop-back.

But only since F# is a *strict* programming language, we have to imitate the laziness explicitly by using the additional level of indirection to create the loop-back.

```
module Block =
    val delay: (unit -> Block<'a, 'b>) -> Block<'a, 'b>
```

Then the call to `Block.transfer` must be defined in the lambda function that we pass in to `Block.delay` like this:

```
let rec b4: Block<int, unit> = Block.delay (fun () -> Blocks.transfer b4)
```

This definition would create an infinite loop, for example. Of course, it is provided for illustrating the idea only.

Also we can add the identity computation to the list of basic definitions for the `Block` computation.

```
module Block =
    val identity<'a> : Block<'a, 'a>
```

The `identity` computation just takes the input transact and immediately returns it within the corresponding `Proc` computation without any delay.

What is important for us is that the `Block` computation might delay the processing of transacts, for the block is based on the `Proc` computation. Therefore, the blocks can work with resources and queues. Also they can have a more complex behavior.

9.2 Transacts

Although the `Block` computation works with generic type parameters, sometimes it is useful to store some context related to the current input transact.

Therefore, IronAivika defines the following data type.

```
[<Sealed; NoEquality; NoComparison>]
type TransactId
and Transact<'a> =
    {
        TransactId: TransactId;
        Value: 'a
    }
```

Then many functions create computations that have the following types:

```
Block<Transact<'a>, Transact<'a>>
Block<Transact<'a>, unit>
```

VisualAivika goes futher and makes these types more specific when working with the transacts:

```
Block<Transact<Map<string, obj>>, Transact<Map<string, obj>>>
Block<Transact<Map<string, obj>>, unit>
```

In other words, VisualAivika can add new named attributes to the transact during the processing by blocks. The `Map` container is immutable. Therefore, it does not affect those transact values that were before adding new attributes. For example, it is important in case of splitting the transacts, which will be considered soon.

To create a new transact, we have to specify the arrival data and some integer priority that will be assigned to the transact. Here it is worth noting that such arrival data are returned by random `Stream` computations.

```
module Transact =
    val create: a:Arrival<'a>
                -> priority:EventPriority
                -> Simulation<Transact<'a>>
```

The value contained in the transact object can be transformed, i.e. the `Transact` data type is a functor.

```
module Transact =
    val map: ('a -> 'b) -> Transact<'a> -> Transact<'b>
```

Nevertheless, even if we transform the transact value then its identifier remains unique for the entire simulation run.

If the transact is created with help of the `Transact.create` function then it becomes belonging to some unique assembly set associated with this transact. We can request the transact for its assembly set instance.

```
type AssemblySet

module Transact =
    val assemblySet: TransactId -> Eventive<AssemblySet>
```

There is another way how new transacts can be created. We can split the existent transact, but then a new clone will belong to the same assembly set which the specified transact belongs to.

```
module Transact =
    val split: Transact<'a> -> Simulation<Transact<'a>>
```

Later, we will return to the subject of assembly sets for transacts.

9.3 Generator Block

Now we know how to create new transacts. The transacts can be created by the arrival data that can be defined by some random stream. Then we can take some terminal chain of blocks and begin processing the transacts by such a block chain.

To formalize this, IronAivika introduces a new data type.

```
type GeneratorBlock<'a> = Block<Block<'a, unit>, unit>
```

The entire processing of transacts becomes after we call the `run` function by the specified terminal block and generator block.

```
module GeneratorBlock =
    val run: Block<'a, unit> -> GeneratorBlock<'a> -> Proc<unit>
```

The result is some action within the `Proc` computation of the corresponding discontinuous process, which must be involved yet in the simulation by using other run functions like `Proc.runInStartTime`.

There are different data constructors for generator blocks. Below are provided only some of them to illustrate the idea.

```
module GeneratorBlock =

    val byStream: EventPriority
                  -> Stream<Arrival<'a>>
                  -> GeneratorBlock<Transact<'a>>

    val bySignal: EventPriority
                  -> Signal<Arrival<'a>>
                  -> GeneratorBlock<Transact<'a>>
```

Here we create a new generator block either by the stream or signal. The first argument specifies the initial transact priorites. There are other data constructors, where the transact priorities can be calculated at time of creation. It is worth noting again that the random `Stream` computations return exactly the same data types that satisfy these function signatures.

If we take the zero priority then we will receive the generator blocks, where new transacts will have the zero priority too.

```
module GeneratorBlock =

    val byStream0: Stream<Arrival<'a>> -> GeneratorBlock<Transact<'a>>
    val bySignal0: Signal<Arrival<'a>> -> GeneratorBlock<Transact<'a>>
```

The latter two functions are just partial applications for the former ones.

9.4 Transact Delay

To simulate some activity, we can hold the corresponding discontinuous process, which can be included in the `Block` computation with help of the following combinator.

```
module Block =
    val within: Proc<unit> -> Block<'a, 'a>
```

For example, if we want to delay every transact by the time interval with uniform distribution betwen (157.0 – 25.0) and (157.0 + 25.0) then we can write:

```
Block.within (Proc.randomUniform_ (157.0 - 25.0) (157.0 + 25.0))
```

On level of implementation, this is just an application of more general `Block.arrc` function.

9.5 Storage

The *storage* is a kind of resource which can be borrowed by few transacts simultaneously. The storage has the initial capacity for the contents, and each transact may consume the arbitrary amount of the contents. In case of deficiency, the transact's computation is blocked until another transact releases the required amount of the contents.

```
[<Sealed>]
type Storage

module Storage =

    val create: capacity:int -> Eventive<Storage>
    val capacity: storage:Storage -> int
    val content: storage:Storage -> Eventive<int>
```

To request the storage for the specified amount of the contents and then release it, we can apply the next functions.

```
module Storage =

    val enter: transact:Transact<'a>
               -> decrement:int
               -> storage:Storage
               -> Proc<unit>

    val leave: increment:int -> storage:Storage -> Eventive<unit>
```

There are more handy wrappers for these functions, where we operate on the `Block` computations. They do the same.

```
module Blocks =

    val inline enter: decrement:int
                      -> Storage
                      -> Block<Transact<'a>, Transact<'a>>

    val inline leave: increment:int
                      -> Storage
                      -> Block<Transact<'a>, Transact<'a>>
```

It is important for us that the `Storage` instance can be returned as a `ResultSource` within simulation. It allows us to observe the statistics and other counters.

9.6 Example: Using Storages

To illustrate the use of `Block` computations, let us take the task, which is a version of task 2E from [6][1].

> At a knitted fabric factory, 50 owned sewing machines operate 8 hours per day, 5 days per week. Any of these machines may fail at any moment. In such a case, it is replaced by a backup machine — either immediately or as soon as one becomes available. Meanwhile, the failed machine is sent to a repair shop, where it is fixed and returned to the workshop — but only as a backup unit.
>
> Operational experience shows that repairing a failed machine takes approximately 7 ± 3 hours, with a uniform distribution. When a machine is in production use, its time-to-failure is uniformly distributed and equals 157 ± 25 hours. The time required to transport a machine between the workshop and the repair shop is negligible and is not taken into account.
>
> The task is to simulate the system's behavior under the assumption that there are 53 sewing machines in total: 50 owned and 3 rented.

The corresponding F# model is stated below.

[1] The task description was translated from Russian into English back.

```
// File: Example2E/Model.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

#nowarn "40"

open System

open Simulation.Aivika
open Simulation.Aivika.Blocks
open Simulation.Aivika.Results

module Model =

    let specs = {

        StartTime=0.0; StopTime=6240.0; DT=1.0;
        Method=RungeKutta4; GeneratorType=SimpleGenerator
    }

    let model = simulation {

        let! men = Storage.create 3 |> Eventive.runInStartTime
        let! nowon = Storage.create 50 |> Eventive.runInStartTime

        let machineStream = Stream.randomUniformInt 0 0
                            |> Stream.take 53

        let rec machines = machineStream |> GeneratorBlock.byStream0

        and chain () = back ()

        and back () =
            Blocks.enter 1 nowon >>>
            Block.within (Proc.randomUniform_ (157.0 - 25.0)
                                              (157.0 + 25.0)) >>>
            Blocks.leave 1 nowon >>>
            Blocks.enter 1 men >>>
            Block.within (Proc.randomUniform_ (7.0 - 3.0)
                                              (7.0 + 3.0)) >>>
            Blocks.leave 1 men >>>
            Block.delay (fun () ->
```

```
            Blocks.transfer (back ()))

    do! machines
            |> GeneratorBlock.run (chain ())
            |> Proc.runInStartTime

    return
        [ResultSource.From ("men", men, "Men");
         ResultSource.From ("nowon", nowon, "NowOn")]
                |> ResultSet.create
}
```

Initially, we create the infinite stream of arrival events at the start time by specifying the delay from the random uniform distribution between 0 and 0, from which we take only 53 items. Thus, we create the stream of arrival events in a count of 53. By that stream, we create the generator block that will activate all the corresponding 53 transacts in the start time of simulation.

The processing of transacts has a loop-back, where all 53 transacts are circulated in the loop again and again. The loop is created with help of the `Block.transfer` function, which is called through the `Block.delay` function that creates the necessary indirection level so that the F# compiler could resolve the definitions.

The following simulation experiment uses the Gtk-based charting back-end. The alternatives are: Windows-based and Skia-based ones.

```
// File: Example2E/RunExperimentGtk.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.Gtk.dll"

#load "Model.fsx"

open Model

open System
open System.IO

open Simulation.Aivika
```

```
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Gtk.Web

let experiment = Experiment ()

experiment.Specs <- Model.specs
experiment.RunCount <- 1000

let men = ResultSet.findByName "men"
let nowon = ResultSet.findByName "nowon"

let menUtil =
  men >> ResultSet.findById StorageContentUtilisationId

let nowonUtil =
  nowon >> ResultSet.findById StorageContentUtilisationId

let provider1 = DeviationChartProvider ()
let provider2 = DeviationChartProvider ()

provider1.Series <- menUtil
provider2.Series <- nowonUtil

let providers =
    [ provider1 :> IExperimentProvider<TextWriter>;
      provider2 :> IExperimentProvider<TextWriter> ]

experiment.RenderHtml (Model.model, providers)
    |> Async.RunSynchronously
```

To run the simulation experiment, I used the following command on Linux[2]:

```
$ dotnet fsi RunExperimentGtk.fsx
Updating directory experiment
Generated file experiment/DeviationChart.png
Generated file experiment/DeviationChart(2).png
Generated file index.html
```

The deviation charts for the content utilization of the both storages are shown on figures 9.1 and 9.2. While the second storage is mostly utilized,

[2]This command is related already to the year 2026, while the previous example from section 1.4 used the old-style command for calling the F# interpreter.

we see that the first storage is less utilized on average than its capacity would allow. The first storage shows how often the repair person men are busy, although with a high deviation from the average.

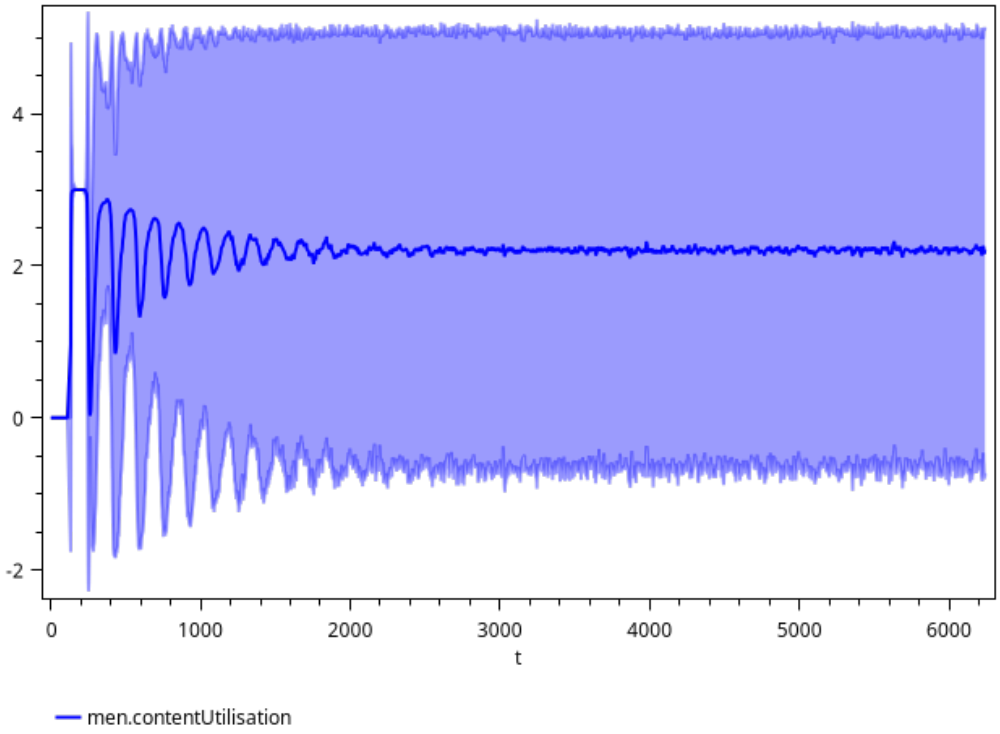


Figure 9.1: The content utilization for the first storage.

9.7 Facility

The facility is such a resource, which can be owned by one transact at time only. But another transact with a higher priority may take the facility by preempting the previous owner. Later, the ownership can be returned to that previous transact. Moreover, that transact can be even redirected to another `Block` computation.

```
type Facility<'a>
```

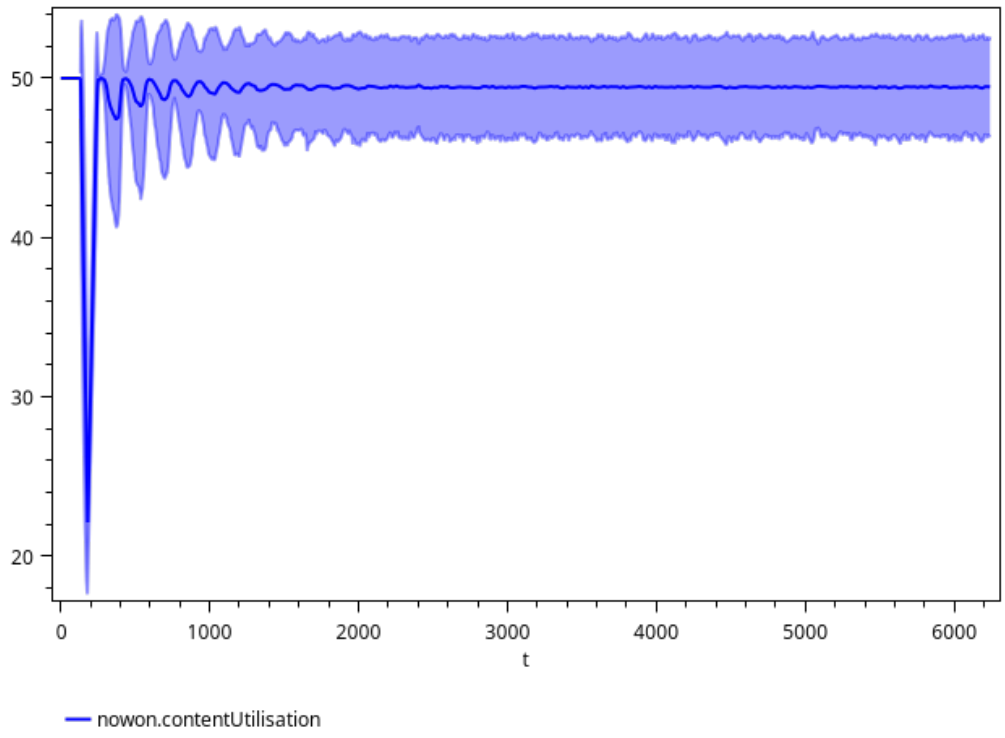


Figure 9.2: The content utilization for the second storage.

```
module Facility =

    val create: unit -> Eventive<Facility<'a>>
    val count: facility:Facility<'a> -> Eventive<int>
```

The next two functions are similar to the analogous ones, which were defined for the `Storage` class type. Here the difference is that the `Facility` has the fixed capacity equaled to 1 only. Either the facility has exactly one owner, or the facility is free.

```
module Facility =

    val seize: Transact<'a> -> Facility<'a> -> Proc<unit>
    val release: Transact<'a> -> Facility<'a> -> Eventive<unit>
```

There are more easy-to-use versions for these functions, which are defined in terms of the `Block` computation already.

```
module Blocks =

    val inline seize: Facility<'a> -> Block<Transact<'a>, Transact<'a>>
    val inline release: Facility<'a> -> Block<Transact<'a>, Transact<'a>>
```

But the main difference between the facility and storage is that the transact may take the ownership and preempt the previous owner if that owner had a less priority.

```
module Facility =

    val preempt: Transact<'a>
                 -> FacilityPreemptMode<'a>
                 -> Facility<'a>
                 -> Proc<unit>

    val return_: Transact<'a> -> Facility<'a> -> Eventive<unit>
```

Before we proceed to the `FacilityPreemptMode` data type description, we provide with the versions which are handy for using them within `Block` computations.

```
module Blocks =

    val inline preempt: PreemptBlockMode<'a>
                        -> Facility<'a>
                        -> Block<Transact<'a>, Transact<'a>>

    val inline return_: Facility<'a>
                        -> Block<Transact<'a>, Transact<'a>>
```

Probably, the facility preemption is one of the most difficult concepts in the `Block` computations. You already saw the resource preemption in section 3.5. The facility preemption is similar, but it can be more difficult, for it has more options.

```
type FacilityPreemptTransfer<'a> =
    Transact<'a> -> float option -> Proc<unit>

type FacilityPreemptMode<'a> =
    {
        PriorityMode: bool;
        Transfer: FacilityPreemptTransfer<'a> option;
```

```
        RemoveMode: bool
    }

type FacilityPreemptMode<'a> with
    static member Default: FacilityPreemptMode<'a>
```

You can find a more detailed description of these options in the VisualAivika documentations, for VisualAivika uses exactly this implementation.

Also there is a more easy-to-use version, which is destined for the `Block` computations directly.

```
type PreemptBlockMode<'a> =
    {
        PriorityMode: bool;
        Transfer: (float option -> Block<Transact<'a>, unit>) option;
        RemoveMode: bool
    }

module PreemptBlockMode =

    val convertTo: PreemptBlockMode<'a> -> FacilityPreemptMode<'a>
    val convertFrom: FacilityPreemptMode<'a> -> PreemptBlockMode<'a>
```

Below the example is provided, where the `PriorityMode` is turned on, but the `RemoveMode` is turned off. It means that the transact with higher priority will take the ownership, but the previous transact that was the owner will not be removed from simulation.

Also the example uses the `Transfer` property, which means that the preempted transact will be redirected to the specified `Block` computation, by saving the ownership, after the preempted transact will return it back. The optional floating-point parameter defines the modeling time interval during which the transact would still have to be held (by some `Proc.hold` computation), until the moment when transact was actually preempted.

To illustrate the example, we need to introduce a concept of simple entity that just updates its counters without any blocking the computations.

As before, the `Facility` instance can be returned as a `ResultSource` from simulation.

9.8 Transact Queue

This is just the entity that can update its counters by request. The counter can be either incremented or decremented. Then we return the corresponding statistics summary.

```
namespace Simulation.Aivika.Blocks

type Queue

module Queue =

    val create: unit -> Eventive<Queue>

    val enqueue: TransactId -> increment:int -> Queue -> Eventive<unit>
    val dequeue: TransactId -> decrement:int -> Queue -> Eventive<unit>
```

Also there are more easy-to-use versions for the `Block` computations. There is no time delay here. They update the counters only.

```
module Blocks =

    val inline queue: increment:int
                      -> Queue
                      -> Block<Transact<'a>, Transact<'a>>

    val inline depart: decrement:int
                       -> Queue
                       -> Block<Transact<'a>, Transact<'a>>
```

Following the general rule, the `Queue` instance can be returned as a `ResultSource` from simulation.

9.9 Example: Facility Preemption

To illustrate the facility preemption, let us take the task, which is a version of the task shown on figure 7.26 from [6][3].

[3]The task description was translated from Russian into English back.

In this problem, the first model segment simulates a closed network of incoming phone calls. Transactions passing through the second model segment simulate students coming for the consultation with the professor. The following assumptions are made:

1. The time unit in the model is 0.01 minutes.
2. The phone rings every 20 ± 5 minutes.
3. The duration of a phone call is exponentially distributed with a mean of 2 minutes.
4. The duration of a conversation with the student is also exponentially distributed with a mean of 10 minutes.
5. When the conversation is interrupted, there is a time surplus of 3 minutes.

It is necessary to estimate the length of the queue of students waiting for a consultation with the professor, under the condition that each new student arrives every 20 ± 5 minutes.

The corresponding F# model is stated below.

```
// File: Example7-26/Model.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

#nowarn "40"

open System

open Simulation.Aivika
open Simulation.Aivika.Blocks
open Simulation.Aivika.Results

type BlockQueue = Simulation.Aivika.Blocks.Queue

module BlockQueue = Simulation.Aivika.Blocks.Queue
```

```
module Model =

    let specs = {

        StartTime=0.0; StopTime=48000.0; DT=1.0;
        Method=RungeKutta4; GeneratorType=SimpleGenerator
    }

    let model = simulation {

        let! line = BlockQueue.create () |> Eventive.runInStartTime
        let! prof = Facility.create () |> Eventive.runInStartTime

        let phoneCallStream = Stream.randomUniform (2000.0 - 500.0)
                                                   (2000.0 + 500.0)

        let studentStream = Stream.randomUniform (2000.0 - 500.0)
                                                 (2000.0 + 500.0)

        let rec phoneCalls = phoneCallStream
                                |> GeneratorBlock.byStream 1

        and phoneCallChain () =
            Block.arrc (fun a -> proc {
                let! f = Facility.isInterrupted prof
                            |> Eventive.lift
                if f then
                    return! Blocks.transfer (busy ())
                                |> Block.run a
                else
                    return a
            }) >>>
            (prof |> Blocks.preempt {
                PriorityMode = true;
                Transfer = Some add;
                RemoveMode = false
            }) >>>
            Block.within (Proc.randomExponential_ 200.0) >>>
            Blocks.return_ prof >>>
            busy ()

        and busy () = Block.terminate

        and students = studentStream
                           |> GeneratorBlock.byStream 0
```

```
        and studentChain () =
            Blocks.queue 1 line >>>
            Blocks.seize prof >>>
            Blocks.depart 1 line >>>
            Block.within (Proc.randomExponential_ 1000.0) >>>
            letGo ()

        and letGo () =
            Blocks.release prof >>>
            Block.terminate

        and add dt =
            let dt = match dt with Some dt -> dt | None -> 0.0
            Block.within (Proc.hold (dt + 300.0)) >>>
            Blocks.transfer (letGo ())

        do! phoneCalls
                |> GeneratorBlock.run (phoneCallChain ())
                |> Proc.runInStartTime

        do! students
                |> GeneratorBlock.run (studentChain ())
                |> Proc.runInStartTime

        return
            [ResultSource.From ("line", line, "Line");
             ResultSource.From ("prof", prof, "Prof")]
                    |> ResultSet.create
    }
```

The model illustrates a few important concepts.

At first, the blocks can be defined recursively like equations, where we use the `Block.trasfer` function to transfer the control flow. If we needed the loop-back then we would apply the `Block.delay` function too.

At second, we can create the `Block` computation by applying the `Block.arrc` function and the `Proc` computation directly, when we need something, which is not provided by the `Blocks` module as a ready-to-use computation.

At third, we can transfer the preempted transact to another block of computations by providing such a transact with the information about the time interval remained.

The `Facility.isInterrupted` computation returns a flag indicating

whether the facility was interrupted. The corresponding computation occurs within `Eventive`, which we lift to the `Proc` computation. Then we decide whether we have to transfer the control to another computation or proceed with the current one. If the facility was interrupted then we transfer the processing of the current transact to the block, which is defined by the `busy` variable.

The following simulation experiment uses the Gtk-based charting back-end. The alternatives are: Windows-based and Skia-based ones.

```
// File: Example7-26/RunExperimentGtk.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.Gtk.dll"

#load "Model.fsx"

open Model

open System
open System.IO

open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Gtk.Web

let experiment = Experiment ()

experiment.Specs <- Model.specs
experiment.RunCount <- 1000

let prof = ResultSet.findByName "prof"

let profQueueLength =
    prof >> ResultSet.findById FacilityQueueCountId

let profQueueLengthStats =
    prof >> ResultSet.findById FacilityQueueCountStatsId
```

```
let provider1 = DeviationChartProvider ()

provider1.Series <-
    [profQueueLength; profQueueLengthStats]
        |> ResultTransform.concat

let providers =
    [ provider1 :> IExperimentProvider<TextWriter> ]

experiment.RenderHtml (Model.model, providers)
    |> Async.RunSynchronously
```

Here we are interested in the length of the queue for students that are awaiting for their consultation with the professor. The corresponding deviation chart is shown on figure 9.3.

Actually, it contains two charts. The chart in blue corresponds to a snapshot of the queue length in the moment, namely, in some discrete time points. The second chart in red shows the statistics, which was collected for the queue length during the simulation. Therefore, the second chart is more smooth. The model is stationary, and the average trends for the both charts must converge if we will increase the final time.

Finally, if you will decide to repeat the simulation experiment then it is highly recommended to create the corresponding .NET project and then compile the code. Then you could significantly increase the number of Monte-Carlo simulation runs.

9.10 Assembly Set

Earlier we introduced the assembly set concept. Every transact created with help of the `Transact.create` function belongs to its own unique `AssemblySet` instance. The default generator blocks use this function. Therefore, their transacts will belong to their own unique assembly sets.

On the contrary, when splitting the transact during simulation with help of the `Transact.split` function, a new clone of the source transact will belong to the same assembly set that becomes shared.

There are two useful functions that allow processing such transacts as the whole, when the transacts are bound to the same assembly set.

```
module Transact =
```

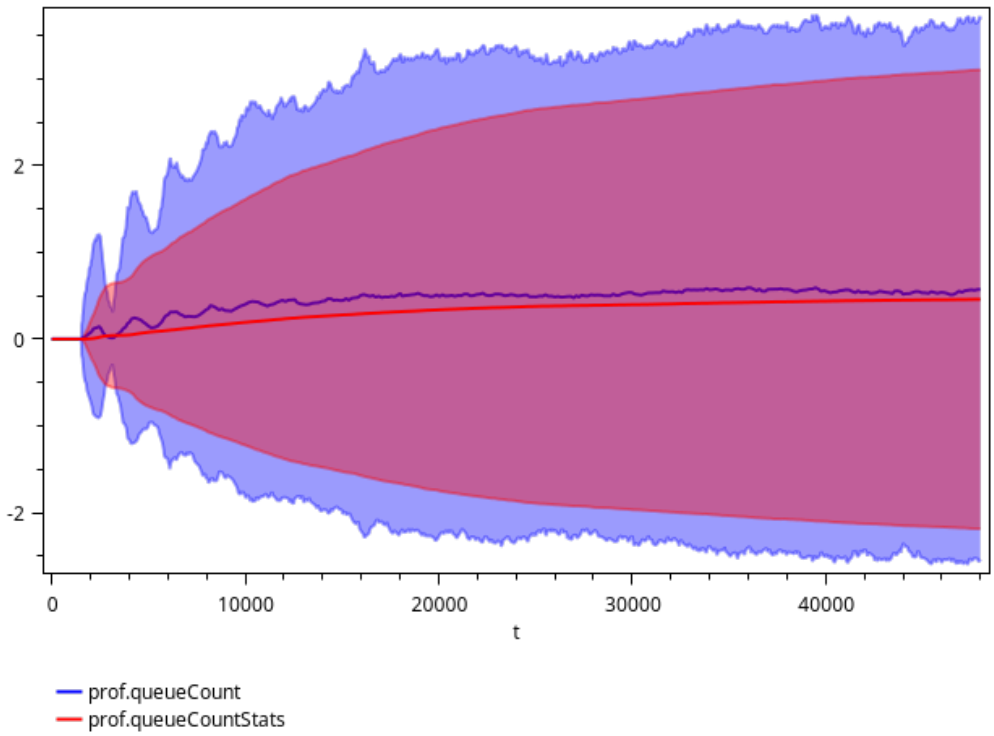


Figure 9.3: The length of the queue for students to be consulted by the professor.

```
    val assemble: TransactId -> count:int -> Proc<unit>
    val gather: TransactId -> count:int -> Proc<unit>
```

They also have versions for the `Block` computation.

```
module Blocks =

    val inline assemble: count:int -> Block<Transact<'a>, Transact<'a>>
    val inline gather: count:int -> Block<Transact<'a>, Transact<'a>>
```

The first function returns a block that assembles the specified number of transacts from the same assembly set in one transact, after they were splitted earlier. But the second function returns a block that delays and gathers the specified number of transacts from the same assembly set,

after they were splitted earlier too. After the transacts are gathered, they continue their execution.

Regarding the splitting of transacts, there are also handly functions for using them within `Block` computations.

```
module Blocks =

    val split: Block<Transact<'a>, unit> list
               -> Block<Transact<'a>, Transact<'a>>

    val inline splitByArray: Block<Transact<'a>, unit> array
                             -> Block<Transact<'a>, Transact<'a>>

    val splitByCount: count:int
                      -> Block<Transact<'a>, unit>
                      -> Block<Transact<'a>, Transact<'a>>
```

The first function splits every incoming transact by the specified list of other blocks which new clones will be transfered to. The second function uses arrays for the same purpose. The third one allows us to specify the common block which the clones will be transfered to, but only we specify the number of clones now.

9.11 Example: Splitting Transacts

To illustrate the transact splitting, let us take the task, which is a version of the task shown on figure 7.31 from [6][4].

> A simple model of a wine factory. Suppose that before labeling a bottle, a worker must seal the filled bottle with a cork. Only one bottle can be sealed at a time. The process requires the worker to place the bottle into a special device that automatically guides the cork into the bottle neck. The operation of sealing a bottle using this semi-automatic method takes 8 ± 3 seconds.
>
> The worker's task is to first seal two dozen bottles, then label them and place them into crates, then seal another two dozen bottles, and so on.

[4]The task description was translated from Russian into English back.

The labeling operation is performed by the same worker as follows: wetting the label, applying it to the bottle, smoothing it out, and then placing the bottle into a crate takes 16±3 seconds (the bottles are pre-filled with wine and already sealed). Each crate holds 12 bottles.

It is necessary to determine the average time the worker spends to pack one crate of wine.

The corresponding F# model is stated below.

```
// File: Example7-31/Model.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"

#nowarn "40"

open System

open Simulation.Aivika
open Simulation.Aivika.Blocks
open Simulation.Aivika.Results

module Model =

    let specs = {

        StartTime=0.0; StopTime=1800.0; DT=1.0;
        Method=RungeKutta4; GeneratorType=SimpleGenerator
    }

    let model = simulation {

        let! workr = Facility.create ()
                        |> Eventive.runInStartTime

        let! stats = Ref.create (SamplingStats.emptyFloats)
                        |> Simulation.lift

        let bottleStream = Stream.randomUniform 0 0
                               |> Stream.take 1
```

```
    let rec bottles = bottleStream |> GeneratorBlock.byStream0

    and bottleChain () =

        Block.arrc (fun input -> proc {
            let! t = Dynamics.time |> Dynamics.lift
            return input |> Transact.map (fun _ -> t)
        }) >>>

        Blocks.seize workr >>>
        Block.within (Proc.randomUniform_ (8.0 - 3.0)
                                          (8.0 + 3.0)) >>>
        Blocks.split [Block.delay (fun () -> bottleChain ())] >>>
        Blocks.release workr >>>
        Blocks.withPriority 1 >>>
        Blocks.gather 24 >>>

        Blocks.seize workr >>>
        Block.within (Proc.randomUniform_ (16.0 - 3.0)
                                          (16.0 + 3.0)) >>>
        Blocks.release workr >>>
        Blocks.assemble 12 >>>

        Block.arrc (fun input -> proc {
            let! t = Dynamics.time |> Dynamics.lift
            let t0 = input.Value
            do! stats |> Ref.modify (SamplingStats.add (t - t0))
                      |> Eventive.lift
            return input
        }) >>>

        Block.terminate

    do! bottles
            |> GeneratorBlock.run (bottleChain ())
            |> Proc.runInStartTime

    return
        [ResultSource.From ("workr", workr, "WORKER");
         ResultSource.From ("stats", stats, "TATYM")]
                |> ResultSet.create
}
```

We start the simulation with one transact only. Then every new transact is permanently cloned with help of splitting.

We gather the statistics summary in a separate mutable `Ref` cell. When receving the initial transact, we put the current modeling time in the transact's value by creating a new instance of the transact, although with the same identifier (i.e. with the same assembly set). The `Transact.map` function does namely this.

When splitting the transact, we apply the `Block.delay` function to introduce a new level of indirection so that the equations could be resolved by the F# compiler.

The process of packing the crates must have a higher priority; otherwise, the model would infinitly simulate the process, when the worker would fill bottles with the cork, which would be wrong.

To set a higher priority, we use the `Blocks.withPriority` function. It was not described earlier, but its behavor should be obvious.

After the crate is packed, we update the corresponding statistics by extracting the time of arrival from the transact's value. It happens within the `Proc` computation.

The following simulation experiment uses the Gtk-based charting backend. The alternatives are: Windows-based and Skia-based ones.

```
// File: Example7-31/RunExperimentGtk.fsx

#I "../../bin"
#r "../../bin/Simulation.Aivika.dll"
#r "../../bin/Simulation.Aivika.Blocks.dll"
#r "../../bin/Simulation.Aivika.Results.dll"
#r "../../bin/Simulation.Aivika.Experiments.dll"
#r "../../bin/Simulation.Aivika.Charting.Gtk.dll"

#load "Model.fsx"

open Model

open System
open System.IO

open Simulation.Aivika
open Simulation.Aivika.Results
open Simulation.Aivika.Experiments
open Simulation.Aivika.Experiments.Web
open Simulation.Aivika.Charting.Gtk.Web

let experiment = Experiment ()
```

```
experiment.Specs <- Model.specs
experiment.RunCount <- 10000

let stats = ResultSet.findByName "stats"

let provider1 = DeviationChartProvider ()

provider1.Series <- stats

let providers =
    [ provider1 :> IExperimentProvider<TextWriter> ]

experiment.RenderHtml (Model.model, providers)
    |> Async.RunSynchronously
```

The deviation chart for the time required for packing one crate of wine is shown on figure 9.4.

9.12 Matching Transacts

Finally, the `MatchChain` type is provided.

```
type MatchChain

module MatchChain =

    val create: unit -> Simulation<MatchChain>
    val matchTransact: TransactId -> MatchChain -> Proc<unit>
```

The `matchTransact` computation waits for until another transact from the same assembly set calls this function.

As usual, there is a more handy version for the `Block` computation.

```
module Block =
    val inline matchTransact: MatchChain
                            -> Block<Transact<'a>, Transact<'a>>
```

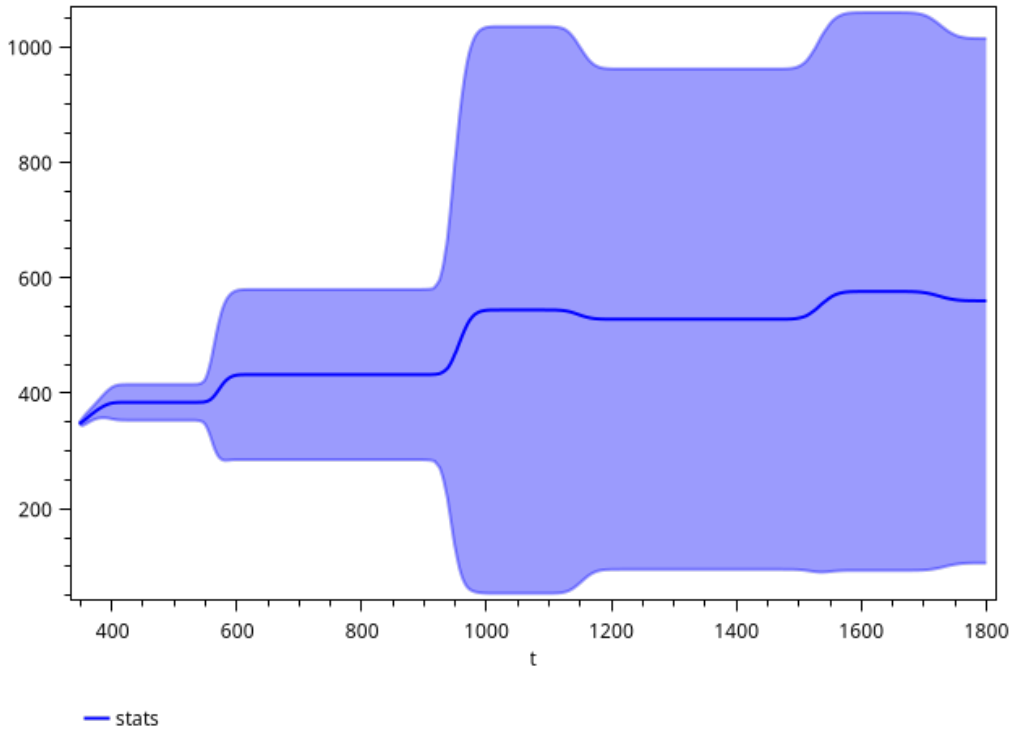


Figure 9.4: The trend and confidence intervals for the process of packing one crate of wine.

Bibliography

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., USA, 1985.

[2] `iThink` Software. `https://www.iseesystems.com`, 2026. Accessed: 10-January-2026.

[3] Robert Macey and George Oster. `Berkeley Madonna` Software. `https://www.berkeleymadonna.com`, 2026. Accessed: 10-January-2026.

[4] Norm Matloff. Introduction to discrete-event simulation and the `SimPy` language. `http://simpy.readthedocs.org/en/latest/`, 2008. Accessed: 1-May-2014.

[5] A.A.B. Pritsker and J.J. O'Reilly. *Simulation with Visual SLAM and AweSim*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1999.

[6] Thomas Schriber. *Simulation using GPSS*. Wiley, 1974.

[7] David E. Sorokin. `Aivika` Library, Version 1.3. `https://hackage.haskell.org/package/aivika`, 2026. Accessed: 10-January-2026.

[8] David E. Sorokin. VisualAivika. `https://visualaivika.ru`, 2026. Accessed: 07-January-2026.

[9] `SimPy` Library. `https://simpy.readthedocs.io/en/latest/`, 2026. Accessed: 10-January-2026.

[10] `AnyLogic` Software. `http://www.anylogic.com`, 2014. Accessed: 1-May-2014.

[11] Ilya I. Trub. *An Object-oriented Modeling in C++*. Piter, Russia, 2006. (In Russian).

[12] `Vensim` Software. `https://vensim.com`, 2026. Accessed: 10-January-2026.