Айвика: имитационное моделирование на языке Haskell

Сорокин Давид Эрнестович <davsor@mail.ru>,
Россия, Марий Эл, Йошкар-Ола

10 сентября 2024 г.

Оглавление

Введение

Я бы хотел, чтобы мои читатели воспринимали эту книгу как приглашение к долгому путешествию в воображаемый мир, где знакомые концепции имитационного моделирования могут быть представлены, возможно, с нового и неожиданного ракурса. Это может добавить в вашу копилку знаний как исследователя еще один подход для создания и имитации моделей, причем такие модели могут быть очень сложными.

Предполагается, что читатель знаком с дискретно-событийным моделированием, и что он может читать код, написанный на языке программирования Haskell. Трудно переоценить роль языка Haskell, поскольку сам метод моделирования, описанный в книге, по всей своей природе глубоко связан с функциональным программированием. Я всегда нахожусь в поиске других подходящих языков программирования, но я все же не смог найти более лучший язык для реализации своих идей.

Книга охватывает последовательное моделирование, описанное в части I, параллельное и распределенное моделирование, которому посвящена следующая часть II, а также вложенное моделирование, затронутое в части III настоящей книги. Все типы моделирования основаны на тех же самых идеях. Более того, многие модели могут быть легко перенесены из одного типа моделирования в другой с небольшими модификациями.

После введения основных моделирующих вычислений в главе 1 книга описывает базовые концепции, примененные в Айвике (*англ.* Aivika)[1]. Так называется программный комплекс моделирования, который описан в настоящей книге, и который я разработал в свое свободное время. Его главные библиотеки распространяются в открытых исходных кодах. Вы можете найти их на Hackage DB[2].

Глава 2 вводит дискретные события и процессы. Это хребет всего комплек-

[1] В слове "Айвика"ударение ставится на последний слог.

[2] Hackage DB имеет следующий адрес: `https://hackage.haskell.org`

са моделирования Айвика. Дискретные процессы определены в терминах вычислений, основанных на так называемых продолжениях. Мы можем использовать такие вычисления как строительные блоки для создания более сложных вычислений. Это ключевое свойство, которое отличает Айвику от многих других платформ и библиотек имитационного моделирования. Оно еще называется композиционностью.

Глава 3 описывает то, как мы можем моделировать ограниченные ресурсы, которые могут быть вытеснены в случае необходимости. Ресурсы - это очень важная концепция в дискретно-событийном моделировании, и они известны под разными названиям. Есть другие очень похожие концепции, описанные в других главах книги.

Глава 4 вводит два небольших типа данных, которые могут быть очень полезны для сбора статистики. Они широко используются в самой Айвике, например, для сбора статистики о поведении очереди или ресурса.

Глава 5 описывает сигналы и задачи. Сигналы широко используются в реализации имитационных экспериментов, к примеру. Но вы также можете использовать сигналы в своих моделях. Они следуют известному из мира программирования шаблону `IObservable`.

Глава 6 посвящена сетям очередей. Она вводит очереди, серверы и два вычисления, которые позволяют соединять разные элементы в одну составную сеть очередей. Такое соединение может быть сделано достаточно декларативно.

Глава 7 вводит основные конструкции, которые могут быть полезны для создания агентных моделей. Она показывает, как мы можем определять агенты, их состояния, а также связанные с ними обработчики тайм-аута и таймера, которые могут задавать достаточно нетривиальное поведение конечного автомата.

Очень маленькая глава 8 посвящена введению в конечные автоматы, определенные в терминах функционального программирования. Она просто описывает, как мы можем моделировать цифровые схемы и некоторые сети.

Глава 9 показывает, как мы можем определять модели системной динамики в терминах комплекса имитационного моделирования Айвика, хотя сам программный комплекс в первую очередь ориентирован на дискретно-событийное моделирование. Тем не менее, мы можем задавать сложные дифференциальные и разностные уравнения, используя массивы в случае необходимости. Также можно задавать уравнения с произвольной задержкой по времени. Более того, эта глава демонстрирует, как мы можем проводить анализ чувствительности по заданным внешним случайным параметрам, используя

метод Монте-Карло, что применимо и для других парадигм моделирования, поддерживаемых в Айвике.

Глава 10 завершает описание методов последовательного моделирования. Она посвящена GPSS-подобному предметно-ориентированному языку, которые очень похож на популярный язык моделирования GPSS. Если вы знакомы с этим языком, то тогда эта глава может быть очень полезна для перевода ваших моделей GPSS на Haskell.

Глава 11 описывает обобщенную версию Айвики. Фактически, обобщенная версия содержит почти тот же код, что включает в себя основная последовательная версия, но первая параметризуется, чтобы подходить для других типов моделирования, таких как распределенное моделирование или вложенное моделирование. Эта глава показывает, как мы можем перевести существующие последовательные модели, превратив их в распределенные модели или в модели вложенного моделирования. Безусловно, такое преобразование не принимает во внимание особенностей того типа моделирования, в который переводим, но мы получаем практически эквивалентную модель.

Глава 12 полностью посвящена параллельному и распределенному моделированию. Она описывает реализацию оптимистичного метода деформации времени. Это позволяет нам создавать распределенные модели большой размерности с высокой степенью уверенности в правильности результатов даже для системы моделирования с откатами вычислений, поскольку в обычном коде на языке Haskell мы можем контролировать побочные эффекты.

Глава 13 вводит в мир вложенного моделирования. Она показывает, как мы можем создавать относительно дешевые и быстрые вложенные модели, ответвляя их от текущей модели. Как предупреждение, мы должны быть очень внимательными с этим типом моделирования, так как дерево ответвлений может иметь экспоненциальную сложность и более. Поэтому мы должны ограничивать дерево ответвлений некоторой глубиной.

Глава 14 описывает другой вид вложенного моделирования, где мы можем использовать дерево особого вида, которое известно под названием решетка. Здесь мы можем создавать вложенные имитации только строго в узлах решетки. Обход узлов решетки имеет уже квадратичную сложность, что делает имитацию и последующую оценку некоторой случайной величины задачей вычислительно достижимой. Например, этот вид вложенного моделирования может быть полезен в финансовом моделировании, но мы должны четко понимать, когда решетка применима, а когда — нет.

Я надеюсь, что вы найдете материал этой книги интересным и полезным для своей учебы и практики. Haskell — красивый язык программирование, и

большинство определений может читаться как строгий математический формализм. По крайней мере, лично я воспринимаю Айвику именно как строгий математический формализм того, как мы можем определять и трактовать многие концепции имитационного моделирования.

Часть I

Последовательное моделирование

В первой части рассматривается последовательное моделирование. Оно охватывает большинство сценариев. Соответствующая реализация в Айвике самая быстрая и самая простая. Здесь запуск имитации однопоточный, а сами операции происходят последовательно друг за другом. Более того, модель одна и не имеет ответвлений.

Другие части рассматривают параллельное и распределенное моделирование, а также вложенное моделирование.

Глава 1

Первое знакомство

Инструкции по установке описаны в приложении А настоящего документа. Вы можете использовать Айвику на трех основных компьютерных платформах: Linux, Windows и macOS. Все примеры, приведенные в этой книге, должны везде работать, возможно используя разные интерфейсы для графиков. Ниже предполагается, что ваш проект зависит от пакетов `aivika`, `aivika-experiment` и `aivika-experiment-chart`.

Прежде, чем мы начнем создавать имитационные модели, мы должны ознакомиться с некоторыми высокоуровневыми концепциями.

1.1 Имитация

В Айвике мы можем трактовать имитацию как полиморфную функцию от запуска:

```
newtype Simulation a = Simulation (Run -> IO a)
```

Использование переменного типа позволяет нам создавать разные сущности внутри имитации. Легко увидеть, что вычисление `Simulation` является монадой.

По заданным параметрам моделирования `Specs` мы можем запустить имитацию, где Айвика создаст объект `Run`, а затем уже запустит вычисление, чтобы получить результат:

```
runSimulation :: Simulation a -> Specs -> IO a
```

Параметры моделирования могут содержать информацию о начальном времени и конечном времени моделирования. Поскольку Айвика также позволяет нам интегрировать дифференциальные уравнения, то мы должны задать шаг интегрирования и метод в независимости от того, будут они использоваться или нет. Также параметры моделирования должны определить генератор случайных чисел, который мы можем использовать в модели.

```
data Specs = Specs { spcStartTime :: Double,
                     -- ^ начальное време
                     spcStopTime :: Double,
                     -- ^ конечное время
                     spcDT :: Double,
                     -- ^ шаг интегрирования
                     spcMethod :: Method,
                     -- ^ метод интегрирования
                     spcGeneratorType :: GeneratorType
                     -- ^ вид генератора случайных чисел
                   }
```

Для простоты мы будем задавать метод Рунге-Кутта 4-го порядка и генератор случайных чисел, предоставляемый по умолчанию.

Например, мы можем задать следующие параметры моделирования:

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

Упомянутый выше тип `Run` довольно сильно зависит от реализации, и поэтому он скрыт внутри API от прямого использования моделистом. По крайней мере, этот тип содержит параметры моделирования, чтобы их можно было передать в каждую часть вычисления `Simulation`.

Чтобы иметь возможность запускать имитации по методу Монте-Карло, значение `Run` также должно содержать информацию о том, как много параллельных прогонов задается в серии, а также информацию о том, какой текущий номер имитации в этой серии. Тогда мы можем параллельно запустить заданное количество имитаций, где каждый запуск будет отличаться своим номером, а также он будет содержать свои собственные экземпляры очереди событий и генератора случайных чисел.

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

Главная идея заключается в том, что многие модели в итоге могут быть сведены к вычислению `Simulation`, какой бы сложной модель ни была. Поэтому для вычислений моделей могут быть тривиальным образом запущены имитации через обозначенные выше функции запуска по заданным параметрам моделирования.

1.2 Внешние параметры

На практике многие модели зависят от внешних параметров, что может быть полезно для проведения анализа чувствительности.

Для представления таких параметров, Айвика использует почти то же самое определение, что мы ранее ввели для представления моделирующего вычисления `Simulation`.

```
newtype Parameter a = Parameter (Run -> IO a)
```

Ключевая разница между двумя вычислениями заключается в том, что параметр может быть *мемоизирован* перед началом имитации так, чтобы итоговое вычисление `Parameter` возвращало постоянную величину внутри каждого запуска имитации, но при этом, чтобы эта величина могла повторно вычисляться и меняться для других запусков (потоко-безопасно).

```
memoParameter :: Parameter a -> IO (Parameter a)
```

Обычно мы должны мемоизировать параметр, если его вычисление не является *чистым* и зависит от таких действий `IO`, как чтение из внешнего файла или генерация случайного числа.

Во время имитации параметры моделирования естественно представлять как вычисления `Parameter`.

```
starttime :: Parameter Double
stoptime :: Parameter Double
dt :: Parameter Double
```

Поскольку в параметрах моделирования мы задаем генератор случайных чисел, то также естественно генерировать случайные числа в рамках вычисления `Parameter`.

```
randomUniform :: Double -> Double -> Parameter Double
randomNormal :: Double -> Double -> Parameter Double
randomExponential :: Double -> Parameter Double
randomErlang :: Double -> Int -> Parameter Double
randomPoisson :: Double -> Parameter Int
randomBinomial :: Double -> Int -> Parameter Int
```

Есть другие встроенные распределения случайных чисел. Пожалуйста, обратитесь к соответствующей документации за более полной информацией.

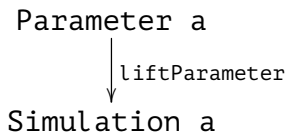
Чтобы поддержать планирование эксперимента, Айвика имеет два дополнительных вычисления, которые просто возвращают соответствующий номер текущего запуска и общее количество запусков в серии, соответственно.

```
simulationIndex :: Parameter Int
simulationCount :: Parameter Int
```

Произвольный параметр может быть преобразован в соответствующее вычисление `Simulation` с помощью следующей функции, которая на самом деле определена в Айвике через класс типов.

```
class ParameterLift m where
  liftParameter :: Parameter a -> m a

instance ParameterLift Simulation
```

Это позволяет использовать параметры внутри имитации.



1.3 Обыкновенные дифференциальные уравнения

В Айвике есть тип `Point` для представления модельной точки времени в рамках текущего запуска имитации. На основе этого типа мы можем задать полиморфную функцию от времени, которая была бы полезной для аппроксимации интегралов.

Соответствующее монадическое вычисление называется `Dynamics`, чтобы подчеркнуть то, что мы можем моделировать некоторый динамический процесс, обычно определяемый через обыкновенные дифференциальные или разностные уравнения системной динамики.

```
newtype Dynamics a = Dynamics (Point -> IO a)
```

Так как модельное время передается в каждую часть вычисления `Dynamics`, то естественно иметь следующее вычисление, которое бы возвращало текущее время.

```
time :: Dynamics Double
```

Существуют разные функции запуска вычисления `Dynamics` в рамках имитации: в начальное время, в конечное время, во всех точках интегрирования, а также в произвольных точках времени по заданным числовым значениям.

```
runDynamicsInStartTime :: Dynamics a -> Simulation a
runDynamicsInStopTime :: Dynamics a -> Simulation a
runDynamicsInIntegTimes :: Dynamics a -> Simulation [IO a]

runDynamicsInTime :: Double -> Dynamics a -> Simulation a
runDynamicsInTimes :: [Double] -> Dynamics a -> Simulation [IO a]
```

1.3.1 Интегралы

Ключевое свойство вычисления `Dynamics` заключается в том, что это вычисление позволяет аппроксимировать интеграл по заданным производной и начальному значению:

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

Второй параметр функции мог бы быть чистым значением, но использование здесь вычисления более удобно для практики, так как позволяет непосредственно ссылаться на начальное значение интеграла при определении дифференциальных уравнений.

Дело в том, что обыкновенные дифференциальные и разностные уравнения могут быть декларативно определены почти так же, как это делается в

математике или во многих коммерческих программных продуктах системной динамики, таких как Vensim[18], ithink/Stella[6], Berkeley-Madonna[7] и Simtegra MapSys[1]

Чтобы создать интеграл, Айвика должна выделить внутренний массив для хранения аппроксимированных значений в точках интегрирования. Такое действие производит побочный эффект в рамках вычисления `Simulation`, где вычисление уже будет иметь доступ к параметрам моделирования.

Кроме этого, Айвика позволяет нам трактовать параметризованный тип `Dynamics` как числовой тип, что значительно упрощает задание дифференциальных и разностных уравнений, что будет продемонстрировано ниже.

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

Например, мы можем переписать модель из руководства "5-минутного введения"в Berkeley-Madonna[7] со следующими уравнениями.

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

Давайте вернем значения интегралов в конечной модельной точке. В то же самое время мы могли бы вернуть значения интегралов в произвольных точках времени, используя другие функции запуска.

```
{-# LANGUAGE RecursiveDo #-}

import Simulation.Aivika
import Simulation.Aivika.SystemDynamics

model :: Simulation [Double]
model =
  mdo a <- integ (- ka * a) 100
      b <- integ (ka * a - kb * b) 0
      c <- integ (kb * b) 0
```

[1] В прошлом автор книги разработал систему визуального моделирования `Simtegra MapSys`, но к сожалению, этот программный продукт был более недоступен для широкой аудитории в момент написания настоящей книги.

```
    let ka = 1
        kb = 1
    runDynamicsInStopTime $ sequence [a, b, c]
```

Здесь мы опираемся на то, что тип `Simulation` является `MonadFix`, а следовательно он поддерживает рекурсивную нотацию-*do*.

Теперь мы можем запустить модель, используя метод Рунге-Кутта 4-го порядка.

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

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

Будь это определено в файле *Main.hs*, мы бы получили следующие результаты моделирования в окне Терминала[2]:

```
$ runghc -package=aivika Main.hs
[2.260329409450236e-4,2.938428231048658e-3,99.99683553882805]
```

Разностные уравнения могут быть определены похожим образом. Читатель может найти пример в дистрибутиве Айвики.

Касательно массивов и векторов интегралов, они создаются естественным образом на Haskell. Никакой специальной поддержки не требуется. Только мы должны будем использовать рекурсивную нотацию-*do* для определения массива, если он имеет обратную связь. Соответствующий пример находится также в дистрибутиве Айвики.

Стоит заметить, что мы можем встраивать внешние функции в дифференциальные уравнения, используя нотацию-*do*. Это возможно благодаря тому, что типы `Simulation` и `Dynamics` являются монадами, впрочем, из-за чего численное интегрирование довольно медленное. Дифференциальные уравнения — далеко не самая сильная сторона Айвики.

[2]Здесь и далее в примерах используется команда `runghc`, что подразумевает, что Айвика установлена с помощью `ghcup`. Те же самые примеры могут быть запущены также с помощью `stack`, но только мы должны были бы создать соответствующий проект Stack.

1.3.2 Мемоизация и побочные эффекты

Существуют вспомогательные функции, которые позволяют нам встраивать в систему уравнений внешние функции, имеющие побочный эффект. Эти вспомогательные функции упорядочивают вычисления в точках интегрирования и используют интерполяцию в других точках времени.

Например, одна из этих функций используется в упомянутой выше функции `integ` для интегрирования.

```
memoDynamics :: Dynamics e -> Simulation (Dynamics e)
memo0Dynamics :: Dynamics e -> Simulation (Dynamics e)
```

Обе функции мемоизируют[3] и упорядочивают вычисление `Dynamics` в точках интегрирования в рамках нового вычисления `Dynamics`, которое возвращают вложенным в другое новое вычисление `Simulation`. Когда запрашивается значение не в точке интегрирования, то обе функции применяют простейшую интерполяцию, которая возвращает значение, вычисленное в ближайшей меньшей точке интегрирования. Однако, обе функции ведут себя по-разному при интегрировании уравнений по методу Рунге-Кутта.

Тип `Point` содержит дополнительную информацию для того, чтобы отличать промежуточные точки интегрирования, используемые методом Рунге-Кутта. Если функция `memoDynamics` мемоизирует значения в этих промежуточных точках времени, то вторая функция `memo0Dynamics` просто игнорирует эти точки, применяя интерполяцию.

Поэтому именно первая мемоизирующая функция `memoDynamics` используется функцией `integ`. Во всех других случаях вторая мемоизирующая функция более предпочтительна, поскольку она более эффективна и потребляет меньше памяти.

Что касается типа `Point`, то он зависит от реализации. Подобно типу `Run` он скрыт от прямого использования моделистом. Определение может измениться в будущем, не оказав эффекта на остальной API.

1.3.3 Стохастические уравнения

Рассмотренные выше моделирующие монады императивны, поскольку они основаны на монаде `IO`. Именно этот факт позволяет использовать генератор случайных чисел во время имитации. Поэтому дифференциальные уравнения

[3] То есть, запоминают значения.

могут быть стохастическими. Айвика предоставляет полезные вспомогательные функции, которые похожи на те, что используются в других программных продуктах системной динамики.

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

memoRandomNormalDynamics ::
  Dynamics Double -> Dynamics Double -> Simulation (Dynamics Double)

memoRandomExponentialDynamics ::
  Dynamics Double -> Simulation (Dynamics Double)

memoRandomErlangDynamics ::
  Dynamics Double -> Dynamics Int -> Simulation (Dynamics Double)

memoRandomPoissonDynamics ::
  Dynamics Double -> Simulation (Dynamics Int)

memoRandomBinomialDynamics ::
  Dynamics Double -> Dynamics Int -> Simulation (Dynamics Int)
```

Они основаны на упомянутых раннее случайных функциях, возвращающих вычисление `Parameter`, но только эти функции мемоизируют сгенерированные значения в точках интегрирования и применяют интерполяцию в других точках времени. Эти функции спроектированы для того, чтобы их использовали в дифференциальных и разностных уравнениях.

1.3.4 Разностные уравнения

Что касается разностных уравнений, то они могут быть построены подобно дифференциальным. Следующая функция по заданной разности и начальному значению возвращает аккумулированную сумму, представленную вычислением `Dynamics`.

```
diffsum :: (Num a, Unboxed a)
           => Dynamics a
           -> Dynamics a
           -> Simulation (Dynamics a)
```

Здесь класс типов `Unboxed` задает, какие типы могут представлять примитивные значения, которые можно более эффективно хранить в памяти. Определение класса можно найти в Айвике.

1.3.5 Уравнения с задержкой

Поскольку аппроксимация интегралов хранится во внутренних массивах, то у Айвики есть возможность запросить значение вычисления `Dynamics` для произвольного прошлого значения модельного времени. Это позволяет создавать дифференциальные уравнения с задержкой.

Основной для создания обратной связи в уравнениях с задержкой является следующая функция:

```
delayIByDT :: Dynamics a
              -> Dynamics Int
              -> Dynamics a
              -> Simulation (Dynamics a)
```

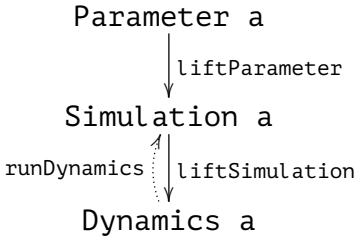
Здесь по заданному первому вычислению, будь то интеграл или какое другое вычисление, заданному множителю шагов интегрирования, на которое мы отступаем в прошлое первого вычисления, и заданному начальному значению создается новое вычисление, которое создает обратную связь для соответствующих вычислений. Мы как бы смотрим в прошлое первого вычисления, а когда не было еще никакого его значения, то используем заданное начальное значение. При этом задержка по времени может быть произвольной и меняться по ходу интегрирования.

1.3.6 Поднятие вычислений

Наконец, мы можем преобразовать произвольное вычисление `Simulation` в соответствующее вычисление `Dynamics`.

```
class SimulationLift m where
  liftSimulation :: Simulation a -> m a

instance SimulationLift Dynamics
instance ParameterLift Dynamics
```

Это буквально означает, что мы можем в обыкновенных дифференциальных и разностных уравнениях использовать внешние параметры, а также вычисления, определенные на уровне запуска имитации:



1.4 Имитационный эксперимент

Помимо моделирующих конструкций, существуют другие вещи, которые желательно иметь реализованными в моделирующем программном комплексе. Чтобы убедиться в правильности модели, или чтобы проанализировать модель, Айвика позволяет нам автоматизировать процесс вывода наиболее важных результатов моделирования.

Моделирующий комплекс может сохранять результаты в файле CSV, который затем может быть открыт в офисном приложении или статистической утилите R для дальнейшего анализа. Также программный комплекс может строить графики и гистограммы по собранной статистике.

Одним из важных графиков является так называемый *график отклонения*, который показывает тренд и доверительные интервалы по правилу *3-х сигм*. Также есть временные графики и графики *XY*, которые Айвика строит для каждого запуска имитации, тогда как график отклонения кумулятивный, и он отображается за весь имитационный эксперимент по методу Монте-Карло, который может состоять из тысяч имитационных запусков.

При запуске имитационного эксперимента Айвика создает локальную веб-страницу, содержащую файл *index.html* и соответствующие вспомогательные файлы. Затем вы можете открыть эту веб-страницу в вашем любимом браузере интернета, чтобы изучить результаты моделирования.

Данный подход в реальности позволяет осуществлять тысячи запусков имитации в рамках одного эксперимента, когда только необходимые данные хранятся в памяти. В то же самое время, браузер интернета становится средством просмотра результатов моделирования.

1.4.1 Возвращение результатов из модели

Во-первых, мы должны подготовить результаты моделирования. Мы связываем или, говоря по другому, ассоциируем каждую переменную с некоторым названием типа `String`, используя функцию `resultSource`. Затем мы собираем такие ассоциации и возвращаем их как одно значение типа `Results` в рамках вычисления `Simulation`.

Наша система обыкновенных дифференциальных уравнений из предыдущего раздела может быть переписана следующим образом.

```
{-# LANGUAGE RecursiveDo #-}

import Simulation.Aivika
import Simulation.Aivika.SystemDynamics
import Simulation.Aivika.Experiment

model :: Simulation Results
model =
  mdo a <- integ (- ka * a) 100
      b <- integ (ka * a - kb * b) 0
      c <- integ (kb * b) 0
      let ka = 1
          kb = 1
      return $
        results
        [resultSource "t" "time" time,
         resultSource "a" "variable A" a,
         resultSource "b" "variable B" b,
         resultSource "c" "variable C" c]
```

Теперь мы можем экспериментировать с этой моделью. Например, мы можем запустить одну имитацию и увидеть результаты моделирования в конечной точке времени.

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

main =
  printSimulationResultsInStopTime
  printResultSourceInEnglish
  model specs
```

Это выведет следующую информацию:

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

-- simulation time
t = 13.0

-- time
t = 13.0

-- variable A
a = 2.260329409450236e-4

-- variable B
b = 2.938428231048658e-3

-- variable C
c = 99.99683553882805
```

Здесь модельное время было напечатано дважды. Библиотечная функция всегда печатает модельное время. Также мы связываем название `t` с модельным временем. Обратите внимание на то, что комментарии разные.

Таким же способом мы могли бы вывести результаты в точках интегрирования. Также мы можем показать информацию либо на русском языке, либо на английском.

Дело в том, что мы можем возвращать переменные разной природы в значении `Results`. Как вы увидите позже, мы можем возвращать ресурсы, очереди, серверы, массивы, списки и т.д. Мы можем настроить это и вернуть свой собственный тип данных в случае необходимости.

Интригующая вещь заключается в том, что значение `Results` на самом деле может быть источником результатов моделирования для разных типов анализа, который может быть большим, чем просто вывод в терминал. Так, мы можем рисовать графики и гистограммы, сохранять результаты в файлах. Давайте посмотрим, как мы можем это сделать.

1.4.2 Задание эксперимента

Теперь мы определяем объект `Experiment` для задания параметров моделирования и числа запусков. Если число запусков более одного, то мы в действительности получаем имитацию по методу Монте-Карло.

```
import Simulation.Aivika.Experiment

experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1,
    experimentTitle = "Chemical Reaction",
    experimentDescription =
      "Chemical Reaction as described in " ++
      "the 5-minute tutorial of Berkeley-Madonna" }
```

Затем мы определяем временные ряды и генераторы, которые уже знают о том, как выводить результаты. Обратите внимание на то, что мы ссылаемся на переменные по их названиям типа `String`, которые мы использовали, когда задавали в модели источники результатов с помощью вызовов функции `resultSource`.

```
t = resultByName "t"
a = resultByName "a"
b = resultByName "b"
c = resultByName "c"

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView $ defaultTableView {
     tableSeries = t <> a <> b <> c,
     tablePredicate =
       do n <- liftDynamics integIteration
          return (n `mod` 10 == 0) },
   outputView $ defaultTimeSeriesView {
     timeSeriesTitle = "Time Series",
     timeSeriesLeftYSeries = a <> b <> c }]
```

Здесь мы задаем то, что хотим сохранить результаты в файле CSV (таблице) и нарисовать график временных рядов. Мы сохраняем каждое 10-е значение в файле CSV, запрашивая текущий номер итерации интегрирования и проверяя, что этот номер делится нацело на 10.

Затем мы выводим веб-страницу по заданным модели и эксперименту, используя наши генераторы. Эта страница будет содержать график и гиперссылки на соответствующий файл CSV.

1.4.3 Вывод графиков

У нас есть выбор. Мы выводим график, используя замечательную библиотеку `Chart`, а следовательно должны выбрать один из интерфейсов графиков. Существует два интерфейса: на основе Cairo и на основе Diagrams. Выбор может зависеть от вашей компьютерной платформы, которая может поддерживать только один из интерфейсов или оба сразу. Пожалуйста, прочтите приложение В настоящего документа для получения большей информации.

Интерфейс графиков на основе Cairo

Используя интерфейс на основе Cairo, мы импортируем необходимые библиотеки и запускаем имитационный эксперимент следующим образом:

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Cairo

import Graphics.Rendering.Chart.Backend.Cairo

import Model
import Experiment

main =
  do let r0 = CairoRenderer PNG
         r  = (WebPageRenderer r0 experimentFilePath)
     runExperimentParallel experiment generators r model
```

Интерфейс графиков на основе Diagrams

Выбрав интерфейс на основе Diagrams, мы импортируем другие библиотеки, а наш запускающий код выглядит несколько иначе.

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Diagrams

import Graphics.Rendering.Chart.Backend.Diagrams

import Model
import Experiment

main =
```

```
  do fonts <- loadCommonFonts
     let r0 = DiagramsRenderer SVG (return fonts)
         r  = WebPageRenderer r0 experimentFilePath
     runExperimentParallel experiment generators r model
```

По заданным модели, эксперименту и генераторам мы запускаем имитационный эксперимент, используя один из интерфейсов графиков.

1.4.4 Запуск имитационного эксперимента

Когда запущен эксперимент с помощью интерфейса на основе Cairo, мы получаем следующий вывод в терминале macOS[4]:

```
$ ghc -O2 -threaded MainUsingCairo.hs
$ ./MainUsingCairo +RTS -N
Updating directory experiment
Generated file experiment/Table(1).csv
Generated file experiment/TimeSeries(1).png
Generated file experiment/index.html
```

Это значит, что приложение создало новый каталог `experiment`, содержащий веб-страницу, которую мы можем открыть в браузере интернета.

Как видно на рисунке 1.1, веб-страница показывает параметры моделирования, временной график, а так же содержит гиперссылку на файл CSV с результатами. График отдельно показан на рисунке 1.2. Мы можем также получить похожие результаты на Windows и Linux, независимо от того, какой бы мы стали использовать интерфейс графиков.

[4]Приведенная команда компиляции была верна для прошлых лет. Уже несколько лет, как нужно дополнительно компилятору GHC задавать названия используемых библиотек через опцию `-package`, если пытаться скомплировать код напрямую как здесь. Сейчас будет проще завести отдельный проект Cabal или Stack.

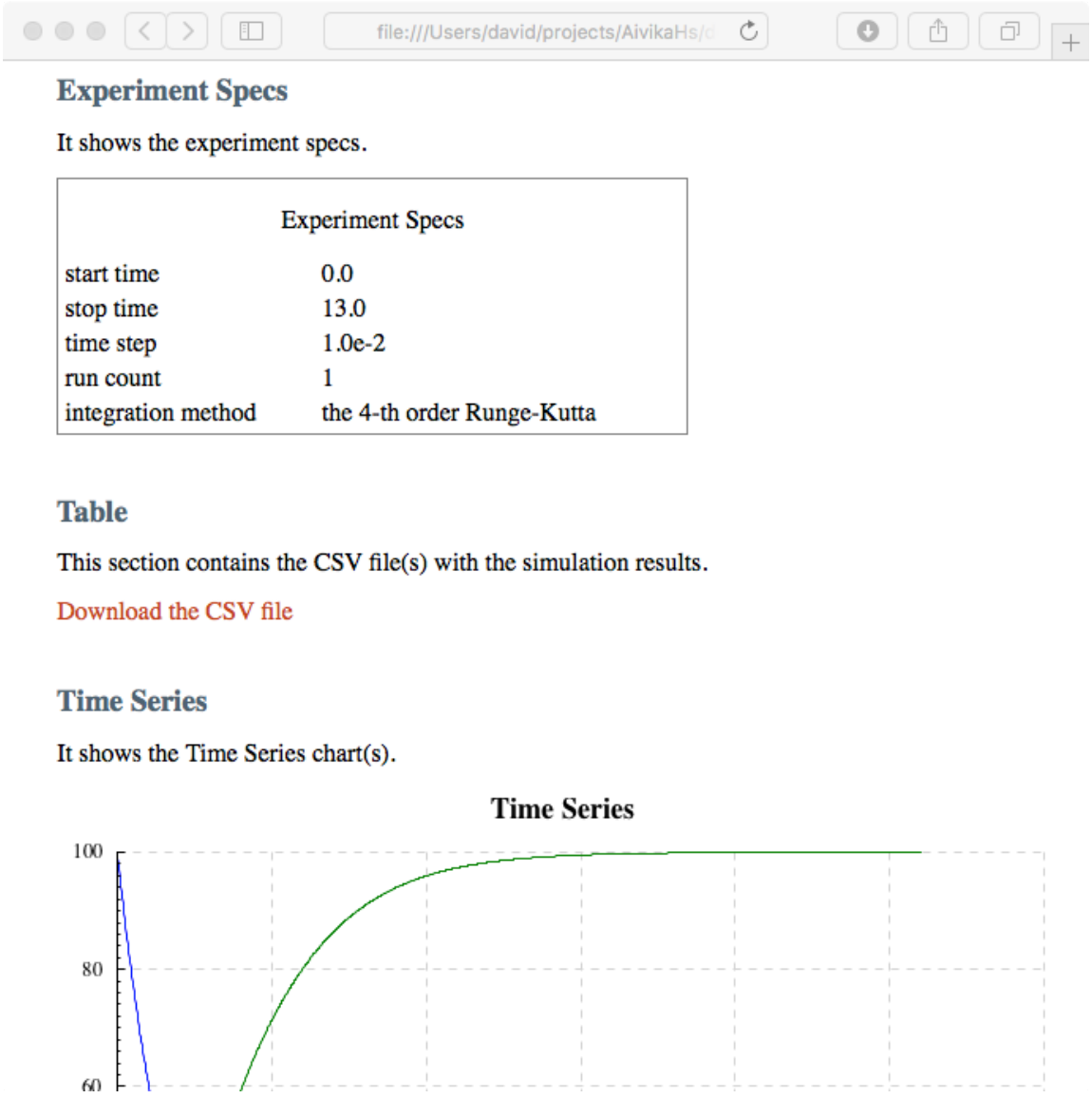


Рис. 1.1: Вывод имитационного эксперимента в браузере интернета.

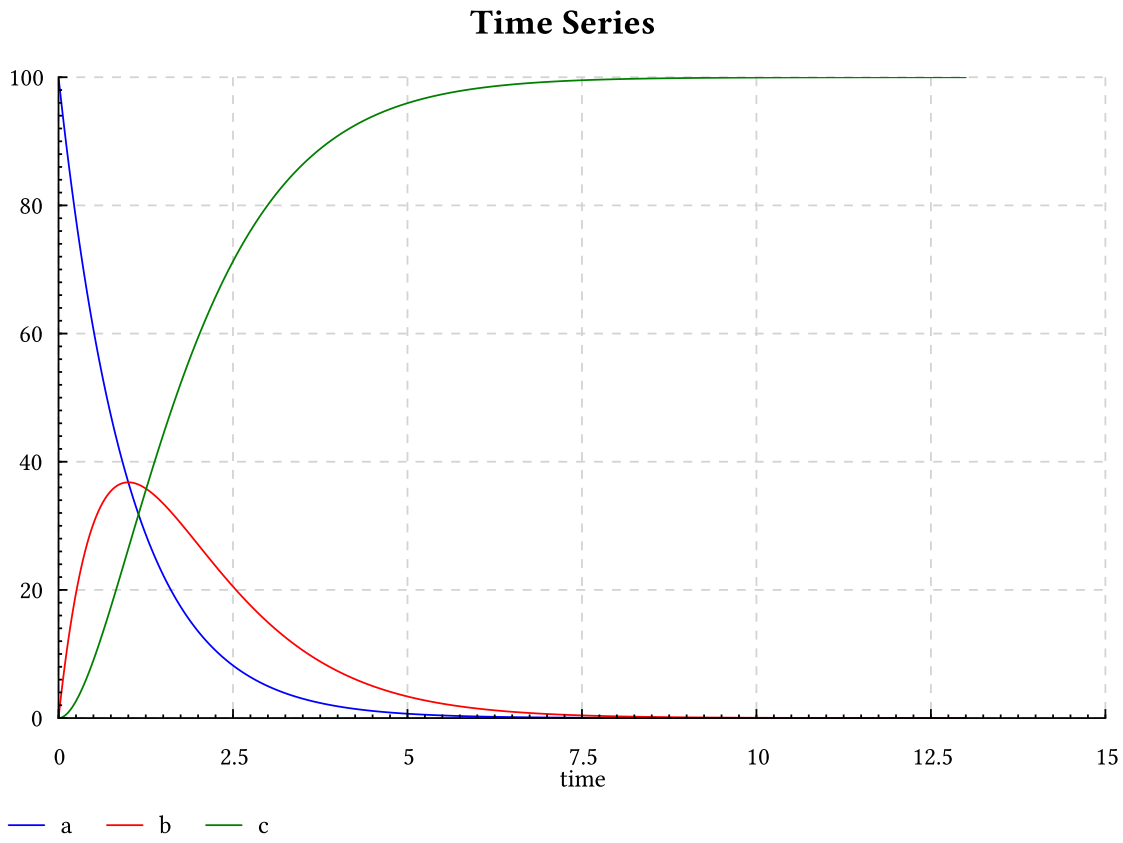


Рис. 1.2: Временные ряды для химической реакции.

Глава 2

Дискретно-событийное моделирование

Ранее мы увидели, как Айвика может быть полезна для интегрирования обыкновенных дифференциальных и разностных уравнений, но Айвика, главным образом, ориентирована на дискретно-событийное моделирование. Данный программный комплекс поддерживает многие парадигмы такого типа моделирования, но их реализация основана на использовании очереди событий. Говоря об Айвике, иначе мы можем сказать, что другие парадигмы дискретно-событийного моделирования в итоге сводятся к событийно-ориентированной парадигме. Это утверждение верно как для процесс-ориентированной парадигмы, так и для парадигмы управляемого временем моделирования, по крайней мере в том смысле, как это представлено в Айвике.

2.1 Событийно-ориентированная парадигма

При *событийно-ориентированной* парадигме[12, 8], мы помещаем все ожидающие обработки события в очередь приоритетов, где самое первое событие будет иметь минимальное время активации. Затем мы последовательно активируем события, удаляя их из очереди. Во время такой активации мы можем добавлять новые события. Эта схема также называется *управляемой событиями*.

Айвика использует почти ту же самую функцию от времени для событийно-ориентированной модели, что мы использовали для аппроксимации интегралов с помощью монады `Dynamics`.

```
newtype Event a = Event (Point -> IO a)
```

Разница заключается в том, что Айвика строго гарантирует[1] на уровне системы типов языка Haskell, что вычисление `Event` всегда будет синхронизировано с очередью событий. Здесь мы подразумеваем, что каждый запуск имитации имеет внутреннюю очередь событий, которая в действительности принадлежит объекту типа `Run`, который мы видели ранее.

Ключевое свойство монады `Event` в том, что она позволяет задать обработчик события, который следует активировать в желаемое модельное время, когда произойдет соответствующее событие.

```
enqueueEvent :: Double -> Event () -> Event ()
```

Чтобы передать сообщение или какие другие данные обработчику события, мы просто используем замыкание при определении обработчика события вторым аргументом.

Отмена событий может быть реализована тривиально. Мы создаем обертку для исходного обработчика события и передаем именно эту обертку функции `enqueueEvent`. Тогда наша обертка уже решает, стоит ли вызывать нижележащий обработчик события или нет. Мы должны обеспечить некоторым механизмом для уведомления этой обертки о том, что исходный обработчик события должен быть отменен. В Айвике есть соответствующая поддержка.

Такой же прием отмены событий может быть адаптирован для реализации обработчиков таймера и тайм-аута в агентном моделировании, как это будет описано позже.

Чтобы быть вовлеченным в имитацию, вычисление `Event` должно быть запущено явно или неявно в рамках вычисления `Dynamics`. Наиболее простая функция приведена ниже. Она активирует все ожидающие выполнения обработчики из очереди событий относительно текущего модельного времени, а затем запускает заданное вычисление.

```
runEvent :: Event a -> Dynamics a
```

Есть тонкий момент касательно вычисления `Dynamics`. В общем, модельное время меняется непредсказуемо внутри `Dynamics`, тогда как внутри вычисления `Event` существует гарантия того, что время синхронизировано с очередью событий, и это время изменяется монотонно.

[1] Хотя на самом деле есть небольшая лазейка в Айвике для обхода этой строгой гарантии.

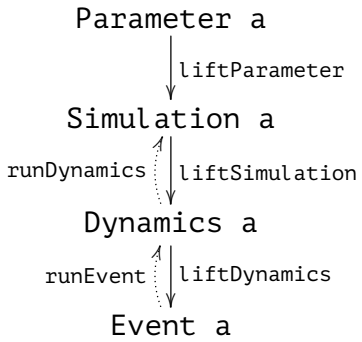
Другие две функции предназначены для наиболее важных сценариев, где мы можем запустить заданное вычисление внутри `Simulation` в начальной или конечной точках модельного времени, соответственно. Эти две функцию уже применяют функцию `runEvent`.

```
runEventInStartTime :: Event a -> Simulation a
runEventInStopTime :: Event a -> Simulation a
```

Следуя правилу, произвольное вычисление `Dynamics` может быть преобразовано в вычисление `Event`. Как и прежде, соответствующая функция определена в классе типов.

```
class DynamicsLift m where
  liftDynamics :: Dynamics a -> m a

instance DynamicsLift Event
instance SimulationLift Event
instance ParameterLift Event
```

Это означает, что интегралы, внешние параметры и вычисления уровня имитационного запуска могут быть непосредственно использованы в событийно-ориентированной модели.



2.2 Изменяемая ссылка

Во многих дискретно-событийных моделях нужны изменяемые ссылки. Поскольку Haskell является языком чистого функционального программирования, то все побочные эффекты должны быть явно выражены в сигнатуре типов. Изменяемые ссылки требуют таких эффектов.

В стандартной библиотеке Haskell есть изменяемая ссылка `IORef`. В Айвике вводится похожий, но только строгий вариант ссылки `Ref`, где все вычисления синхронизированы с очередью событий.

```
data Ref a

newRef :: a -> Simulation (Ref a)

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

В имитационных моделях в рамках вычисления `Simulation` следует использовать тип `Ref` вместо `IORef` там, где это только возможно.

Особенно это важно для распределенного моделирования, которое также поддерживается Айвикой. По причине того, что любое действие `IO` приводит к синхронизации глобального виртуального времени среди всех логических процессов, вовлеченных в распределенную имитацию. В то же время, тип `Ref` гораздо эффективнее, поскольку он поддерживает откаты, и поэтому нет необходимости в тяжеловесной синхронизации.

Что касается вложенного моделирования, также поддерживаемого Айвикой, то тип `Ref` имеет дополнительное свойство. Такая ссылка может быть эффективно изменена в производной вложенной ветви имитации без какого-либо воздействия на значение ссылки в предшествующих ветвях той же самой имитационной модели. Но давайте вернемся к описанию основ Айвики.

2.3 Пример: событийно-ориентированная модель

Дистрибутив Айвики содержит примеры использования изменяемых ссылок в дискретно-событийных моделях. Одна из таких моделей приведена ниже. Сама задача описана в документации к SimPy[8].

> Есть два станка, которые иногда ломаются. Время наработки распределено экспоненциально со средним 1.0, и время восстановления распределено экспоненциально со средним 0.5. Есть двое мастеров. Так что, оба станка могут быть отремонтированы одновременно, если они сломались в то же время. Вывод задает долговременную пропорцию времени наработки. Должны получить значение около 0.66.

Ниже приведена соответствующая модель. Стоит заметить, что представленное определение не является самым простым. Как мы увидим в разделе 2.6, существуют другие способы. Только нам нужно ввести некоторые новые концепции, прежде чем мы могли бы переписать модель.

```
import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
meanRepairTime = 0.5

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

model :: Simulation Results
model =
  do totalUpTime <- newRef 0.0

     let machineBroken :: Double -> Event ()
         machineBroken startUpTime =

           do finishUpTime <- liftDynamics time
              modifyRef totalUpTime (+ (finishUpTime - startUpTime))
              repairTime <-
                liftParameter $
                randomExponential meanRepairTime

              -- положить в очередь новое событие
              let t = finishUpTime + repairTime
              enqueueEvent t machineRepaired

         machineRepaired :: Event ()
         machineRepaired =

           do startUpTime <- liftDynamics time
              upTime <-
                liftParameter $
                randomExponential meanUpTime

              -- положить в очередь новое событие
              let t = startUpTime + upTime
              enqueueEvent t $ machineBroken startUpTime
```

```
      runEventInStartTime $
        do -- запустить первый станок
           machineRepaired
           -- запустить второй станок
           machineRepaired

      let upTimeProp =
            do x <- readRef totalUpTime
               y <- liftDynamics time
               return $ x / (2 * y)

      return $
        results
        [resultSource
         "upTimeProp"
         "The long-run proportion of up time (~ 0.66)"
         upTimeProp]

main =
  printSimulationResultsInStopTime
  printResultSourceInEnglish
  model specs
```

Мы моделируем два станка, каждый из которых может находиться в одном из двух состояний. Каждое состояние представлено как вычисление `Event`. Эти вычисления могут быть зарегистрированы как обработчики событий. Активируя обработчик, мы меняем состояние станка.

После запуска примера мы получаем желаемый результат с некоторой точностью.

```
$ runghc -package=aivika -package=mtl MachRep1EventDriven.hs
----------

-- simulation time
t = 1000.0

-- The long-run proportion of up time (~ 0.66)
upTimeProp = 0.6635359777536585
```

Честно говоря, использование событийно-ориентированной парадигмы может показаться довольно громоздким. Айвика поддерживает более высокоуровневые парадигмы. Позже будет показано, как та же самая задача может быть решена более изящным и простым способом.

2.4 Переменная с памятью

Иногда нам нужно смешивать обыкновенные дифференциальные уравнения с дискретно-событийной моделью. В большинстве случае было бы более простым и эффективным закодировать метод Эйлера непосредственно в рамках вычисления `Event`. Однако, если вам все же нужны сложные дифференциальные уравнения, определенные в рамках вычисления `Dynamics`, то тогда существует метод, который позволяет создавать комбинированные имитационные модели. Он небесплатен и имеет свою стоимость.

Для этого существует аналог изменяемой ссылки, который хранит историю своих значений. Айвика определяет соответствующий тип `Var`. Он имеет почти такие же функции с похожей сигнатурой, что имеет тип `Ref`.

```
data Var a

newVar :: a -> Simulation (Var a)

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

Тем не менее, мы также можем использовать переменную в дифференциальных и разностных уравнениях, запрашивая *первое* актуальное значение для *каждой* точки времени с помощью следующей функции, которая активирует все ожидающие исполнения события, если это необходимо.

```
varMemo :: Var a -> Dynamics a
```

Магия заключается в следующем. Переменная `Var` хранит историю изменений. При обновлении переменной или при запросе значения в новой точке времени, объект данных `Var` сохраняет внутри себя значение, которое было первым для запрашиваемой точки времени. Затем оно становится постоянным в течение имитации. Поэтому возвращаемое функцией `varMemo` вычисление может быть использовано в дифференциальных и разностных уравнениях системной динамики.

Напротив, функция `readVar` возвращает вычисление *последнего* актуального значения для *текущей* точки модельного времени. Это значение уже предназначено для того, чтобы его использовали в рамках дискретно-событийной модели, поскольку оно синхронизировано с очередью событий. Такое вычисление `Event` просто обязано быть синхронизировано с очередью событий.

В случае необходимости мы можем временно "заморозить"переменную и вернуть ее внутреннее состояние: тройки из времени, первого и последнего значений для соответствующего значения времени.

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

Возвращаемые этой функцией значения времени уникальны и отсортированы по возрастанию.

Стоит заметить, что переменная `Var` медленная. Было бы логической ошибкой использовать `Var` только для сбора статистики. Это было бы довольно неэффективно. Более того, это потребляло бы много памяти. В Айвике есть специальные структуры данных, которые следует использовать для сбора статистики. Смотрите разделы 4.1 и 4.2 для получения дополнительной информации.

2.5 Процесс-ориентированная парадигма

При *процесс-ориентированной* парадигме[12, 8] мы моделируем имитационные активности с помощью особого вида процессов. Мы можем явно приостанавливать и возобновлять такие процессы. Также мы можем запрашивать и освобождать ресурсы, неявно приостанавливая и возобновляя процессы в случае необходимости.

Айвика на самом деле поддерживает процесс-ориентированное моделирование на разных уровнях. Так, существуют потоки данных, а также GPSS-подобные блоки. В этом разделе описан нижний уровень, который, тем не менее, является основанием для более высоких уровней.

2.5.1 Дискретные процессы

Дискретный процесс представлен монадой `Process`, основанной на продолжениях. Более того, существует связанный с ним тип идентификатора процесса `ProcessId`.

```
data Process a
data ProcessId

newProcessId :: Simulation ProcessId
```

Мы можем запустить процесс в рамках имитации с помощью одной из следующих функций.

```
runProcess :: Process () -> Event ()
runProcessUsingId :: ProcessId -> Process () -> Event ()

runProcessInStartTime :: Process () -> Simulation ()
runProcessInStartTimeUsingId :: ProcessId -> Process () -> Simulation ()

runProcessInStopTime :: Process () -> Simulation ()
runProcessInStopTimeUsingId :: ProcessId -> Process () -> Simulation ()
```

Если идентификатор процесса не задан, то создается новый, и он назначается во время запуска процесса. Каждый процесс всегда имеет свой собственный уникальный идентификатор.

```
processId :: Process ProcessId
```

Характерным свойством монады `Process` является то, что процесс может быть приостановлен на заданный промежуток модельного времени через очередь событий.

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

Это позволяет моделировать некоторые активности, задерживая вычисление. После истечения промежутка времени вычисление возобновится с другой точки модельного времени равной сумме текущего модельного времени и заданного промежутка времени за исключением оговорок, изложенных ниже.

Такой приостановленный процесс может быть немедленно возобновлен, и мы можем узнать, действительно ли приостановка была прервана. Информация об этом хранится до следующего вызова функции `holdProcess`.

```
interruptProcess :: ProcessId -> Event ()
processInterrupted :: ProcessId -> Event Bool
```

Если приостановка была прервана таким способом, то процесс продолжает свое выполнение, как если бы действие `holdProcess` закончилось бы немедленно. Но есть также другой способ прерывания, который называется *вытеснением*. Вытесненный процесс временно приостанавливается и затем может возобновить выполнение действия `holdProcess` с того места, где был

вытеснен. Пожалуйста, посмотрите раздел 3.8 для получения дополнительной информации.

Стоит обратить внимание на типы вычислений, которые возвращаются функциями. Тип результата `Event` означает, что вычисление выполняется немедленно, и оно не может быть прервано. Напротив, тип результата `Process` значит, что соответствующее вычисление может быть приостановлено, даже навсегда. Очень важно это понимать.

Мы можем использовать следующие функции, чтобы "усыпить"процесс на неопределенное время, так чтобы его могли потом "разбудить".

```
passivateProcess :: Process ()
processPassive :: ProcessId -> Event Bool
reactivateProcess :: ProcessId -> Event ()
```

Каждый процесс может быть немедленно отменен, что важно для моделирования некоторых активностей.

```
cancelProcessWithId :: ProcessId -> Event ()
cancelProcess :: Process a
processCancelled :: ProcessId -> Event Bool
```

Иногда бывает нам нужно запустить произвольное подвычисление с заданным тайм-аутом, но имейте в виду, что это очень медленное действие.

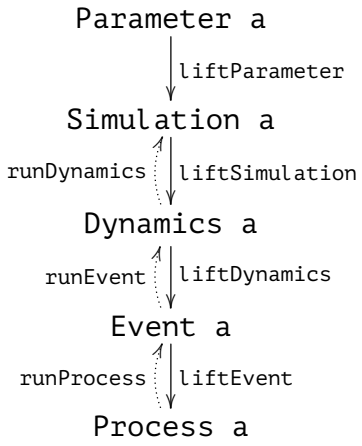
```
timeoutProcess :: Double -> Process a -> Process (Maybe a)
```

Если подпроцесс выполняется слишком долго и выходит за пределы выделенного временного промежутка, то тогда такой подпроцесс немедленно отменяется, и возвращается `Nothing` в рамках вычисления `Process`. Иначе вычисленное значение вернется сразу после того, как будет получено подпроцессом.

Каждое рассмотренное нами ранее моделирующее вычисление может быть преобразовано в вычисление `Process`.

```
class EventLift m where
  liftEvent :: Event a -> m a

instance EventLift Process
instance DynamicsLift Process
instance SimulationLift Process
instance ParameterLift Process
```

Это позволяет использовать интегралы и внешние параметры, а также обновлять изменяемые ссылки и переменные с памятью в рамках уже процесс-ориентированной имитации. Все это позволяет комбинировать как событийно-ориентированные, так и процесс-ориентированные части одной модели.



2.5.2 Создание параллельных процессов

Другой процесс может быть запущен параллельно на-лету. Если тот второй процесс никак не связан с текущим родительским процессом, то тогда мы можем запустить его в рамках вычисления `Event`, а затем преобразовать результат в вычисление `Process`. Нет необходимости в дополнительной функции. Достаточно иметь `liftEvent` и одну из функций запуска для `Process`.

```
liftEvent $ runProcess (p :: Process ())
```

Однако, если жизненный цикл дочернего процесса должен быть привязан к жизненному циклу родительского процесса, так чтобы эти процессы могли быть отменены в некотором порядке в случае необходимости, то тогда нам следует использовать одну из следующих функций.

```
spawnProcess :: Process () -> Process ()
spawnProcess = spawnProcessWith CancelTogether

spawnProcessWith :: ContCancellation -> Process () -> Process ()
```

Здесь первый аргумент второй функции задает, как связаны оба процесса.

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

Упомянутая выше функция `timeoutProcess` как раз использует функцию `spawnProcessWith`, чтобы запустить заданный подпроцесс в пределах таймаута.

Также произвольное число вычислений `Process` может быть запущено параллельно, где мы будем ожидать завершения всех подпроцессов, чтобы вернуть окончательный результат.

```
processParallel :: [Process a] -> Process [a]
```

2.5.3 Мемоизация

Вычисление `Process` может быть мемоизировано так, чтобы результирующий процесс всегда возвращал то же самое значение в течение имитации независимо от того, как много раз результирующий процесс будет вызван повторно. Исходный процесс вызывается только один раз.

```
memoProcess :: Process a -> Simulation (Process a)
```

Иногда это может быть полезно, например, если мы собираемся клонировать поток данных.

2.5.4 Обработка исключительных ситуаций

Известно, что продолжения, на основе которых реализована монада дискретных процессов `Process`, трудны для правильной обработки исключительных ситуаций. Тем не менее, автор Айвики успешно адаптировал подход F# `Async` и добавил соответствующие функции, чтобы обрабатывать исключения в рамках `IO`.

```
catchProcess :: Exception e => Process a -> (e -> Process a) -> Process a
finallyProcess :: Process a -> Process b -> Process a
throwProcess :: IOException -> Process a
```

Существуют похожие функции для обработки исключительных ситуаций в рамках всех других моделирующих монад, рассмотренных ранее в настоящем документе.

Стоит упомянуть, что вычисление `Process` выполняется быстрее, пока не включена обработка исключительных ситуаций. Как только вы примените функцию `catchProcess` или `finallyProcess`, то будут включены дополнительные проверки, из-за которых имитация станет медленнее.

2.5.5 Случайные задержки процессов

В моделях нам часто нужно задерживать процессы на случайные интервалы времени по заданным распределениям. Айвика определяет соответствующие вспомогательные функции. Все они порождают временной промежуток, а затем передают его значение функции `holdProcess`, как показано ниже.

```
randomUniformProcess min max =
  do t <- liftParameter $ randomUniform min max
     holdProcess t
     return t
```

Для каждого распределения существуют две близкие функции. Первая выполняет действие и возвращает значение использованного промежутка времени. Вторая функция просто выполняет действие. Ниже приведены некоторые из функций. Существуют похожие функции и для других реализованных распределений.

```
randomUniformProcess :: Double -> Double -> Process Double
randomUniformProcess_ :: Double -> Double -> Process ()

randomNormalProcess :: Double -> Double -> Process Double
randomNormalProcess_ :: Double -> Double -> Process ()

randomExponentialProcess :: Double -> Process Double
randomExponentialProcess_ :: Double -> Process ()

randomErlangProcess :: Double -> Int -> Process Double
randomErlangProcess_ :: Double -> Int -> Process ()

randomPoissonProcess :: Double -> Process Int
randomPoissonProcess_ :: Double -> Process ()

randomBinomialProcess :: Double -> Int -> Process Int
randomBinomialProcess_ :: Double -> Int -> Process ()
```

Таким образом, описанные в этом разделе функции позволяют эффективно моделировать довольно сложные активности. Тем не менее, вычисление `Process` — это не самый высокий уровень абстракции, хотя и один из самых удобных. Айвика поддерживает более высокоуровневые вычисления, описанные далее.

2.6 Пример: процесс-ориентированная модель

Давайте вернемся к нашей задаче, которую мы решили в разделе 2.3, используя событийно-ориентированную парадигму. Ниже приведена повторно постановка задачи. Она соответствует документации к SimPy.

> Есть два станка, которые иногда ломаются. Время наработки распределено экспоненциально со средним 1.0, и время восстановления распределено экспоненциально со средним 0.5. Есть двое мастеров. Так что, оба станка могут быть отремонтированы одновременно, если они сломались в то же время. Вывод задает долговременную пропорцию времени наработки. Должны получить значение около 0.66.

2.6.1 Возвращение результатов из модели

Используя процессы, мы можем решить задачу более изящно. Во-первых, мы должны написать модель, которая бы возвращала результаты моделирования.

```
module Model(model) where

import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
meanRepairTime = 0.5

model :: Simulation Results
model =
  do totalUpTime <- newRef 0.0

     let machine :: Process ()
         machine =
```

```
        do upTime <-
             randomExponentialProcess meanUpTime
           liftEvent $
             modifyRef totalUpTime (+ upTime)
           repairTime <-
             randomExponentialProcess meanRepairTime
           machine

  runProcessInStartTime machine
  runProcessInStartTime machine

  let upTimeProp =
        do x <- readRef totalUpTime
           y <- liftDynamics time
           return $ x / (2 * y)

  return $
    results
    [resultSource
     "upTimeProp"
     "The long-run proportion of up time (~ 0.66)"
     upTimeProp]
```

Читатель может сравнить эту модель с предыдущей. Концептуально они делают одно и тоже: используют ту же самую очередь событий и имеют одно и то же поведение.

Также мы можем проверить, что модель ведет себя ожидаемо, если добавим трассирующие сообщения, которые описаны в приложении C.

Возвращаясь к текущей модели, мы могли бы запустить имитацию и вывести результаты непосредственно на терминал, но было бы более интересным показать, как мы можем запустить имитацию по методу Монте-Карло, чтобы нарисовать график отклонения с доверительными интервалами и нарисовать гистограмму.

2.6.2 Задание эксперимента

Мы определяем эксперимент с 1000-ю имитационными запусками. Параметры моделирования те же, что и ранее.

```
module Experiment (experiment, generators) where

import Data.Monoid
```

```
import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

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

experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1000,
    experimentTitle = "MachRep1 Example",
    experimentDescription = "The simulation experiment." }

x = resultByName "upTimeProp"

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView defaultInfoView,
   outputView $ defaultDeviationChartView {
     deviationChartPlotTitle = "The Up-time Proportion Chart",
     deviationChartLeftYSeries = x },
   outputView $ defaultFinalHistogramView {
     finalHistogramPlotTitle = "The Up-time Proportion Histogram",
     finalHistogramSeries = x }]
```

Обратите внимание на то, как мы ссылаемся на переменные по их названию. Модель декомпозирует моделируемые сущности так, чтобы мы могли вернуть их единообразно как некоторое значение типа `Results`. Поэтому мы должны восстановить переменные по их названиям.

Здесь мы показываем параметры эксперимента, показываем информацию по переменным, рисуем график отклонения и затем показываем гистограмму по значениям, собранным в конечных точках времени для всех имитационных запусков.

2.6.3 Вывод графиков

Применяя интерфейс графиков на основе Cairo, мы запускаем имитационный эксперимент с помощью следующего кода.

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Cairo

import Graphics.Rendering.Chart.Backend.Cairo

import Model
import Experiment

main =
  do let r0 = CairoRenderer PNG
         r  = (WebPageRenderer r0 experimentFilePath)
     runExperimentParallel experiment generators r model
```

Напротив, мы могли бы запустить эксперимент, используя интерфейс графиков на основе Diagrams. Тогда мы бы получили файлы рисунков в формате SVG вместо PNG, как это должно работать на платформе Windows.

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Diagrams

import Graphics.Rendering.Chart.Backend.Diagrams

import Model
import Experiment

main =
  do fonts <- loadCommonFonts
     let r0 = DiagramsRenderer SVG (return fonts)
         r  = WebPageRenderer r0 experimentFilePath
     runExperimentParallel experiment generators r model
```

Вы могли заметить, что код такой же как и тот, что был определен в разделе 1.4.3. Поэтому он более не будет повторяться.

2.6.4 Запуск имитационного эксперимента

Результаты в виде графика отклонения и гистограммы показаны на рисунках 2.1 и 2.2, соответственно. На моем не очень новом Macbook Pro имитация по методу Монте-Карло с 1000-ю (тысячью) запусков длилась 2.847 секунды (менее трех секунд). Это включает и создание графиков.

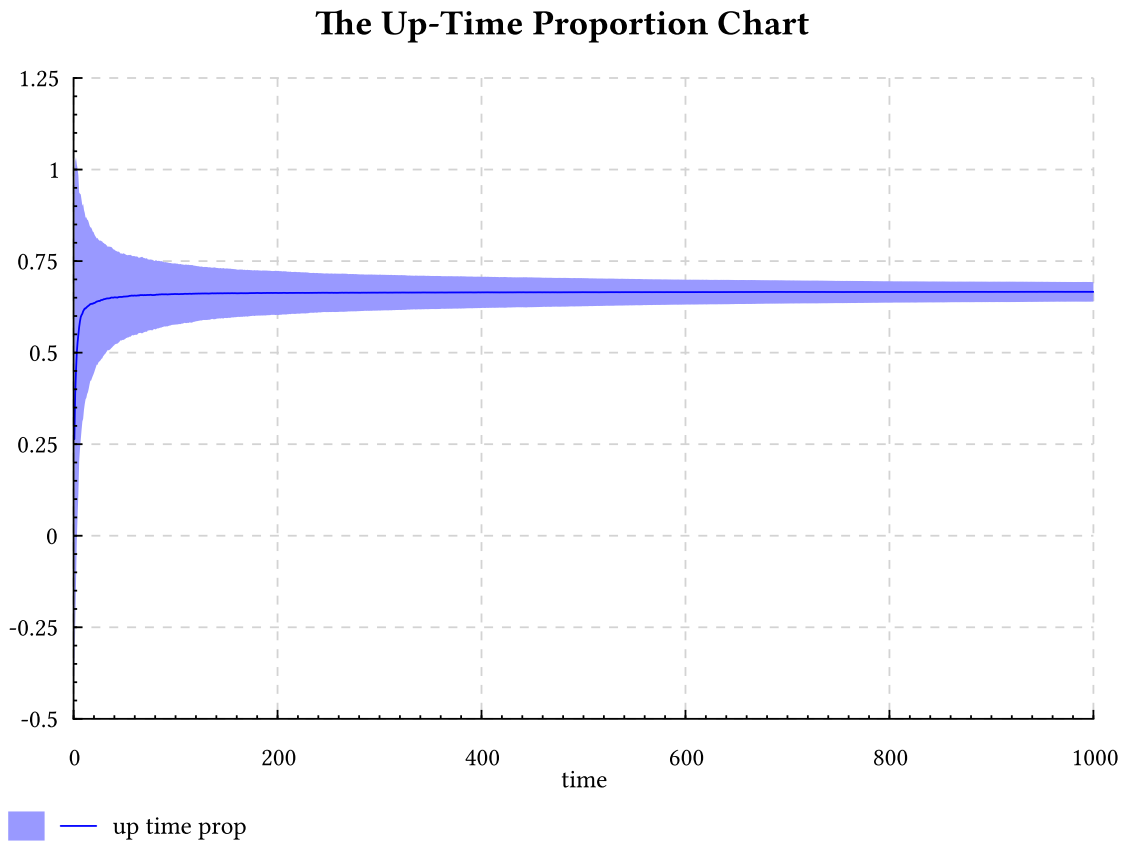


Рис. 2.1: График отклонения для долговременной пропорции времени наработки.

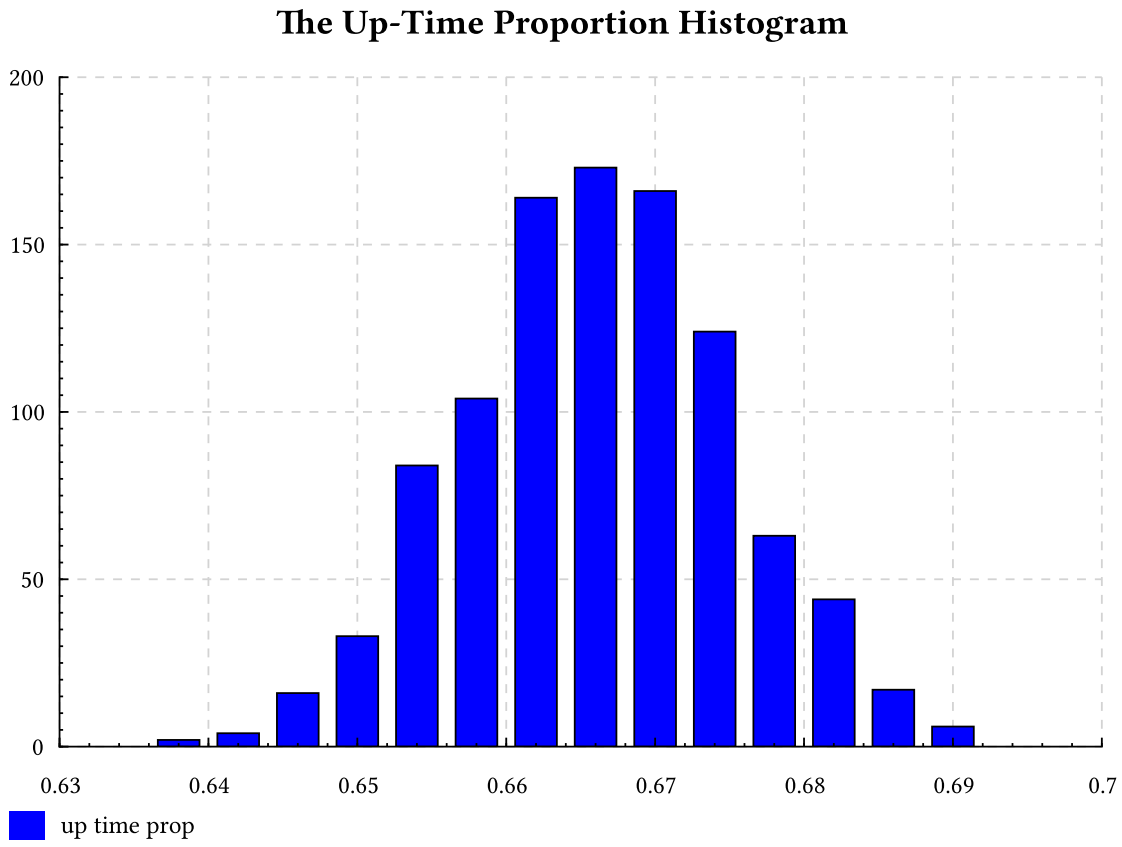


Рис. 2.2: Гистограмма долговременной пропорции времени наработки.

Существует также другая популярная парадигма, которая может быть применена к дискретно-событийному моделированию. Она обычно дает более грубые результаты, так как мы должны уже будем масштабировать модельное время. Следующие два раздела показывают, как Айвика поддерживает ту парадигму, которую мы можем применить для решения той же самой задачи.

2.7 Управляемое временем моделирование

При *управляемой временем*[2] парадигме[12, 8] дискретно-событийного моделирования мы разбиваем время на крошечные интервалы. В каждой точке времени мы просматриваем все активности и проверяем возможное выполнение событий.

Идея заключается в том, что мы можем естественным образом представить активность как вычисление `Event`, которое мы будем периодически вызывать через очередь событий[13].

```
enqueueEventWithTimes :: [Double] -> Event () -> Event ()
enqueueEventWithTimes ts e = loop ts
  where loop []       = return ()
        loop (t : ts) = enqueueEvent t $ e >> loop ts
```

Также мы можем использовать другую предопределенную функцию, которая делает почти то же самое, но только она вызывает заданное вычисление непосредственно в точках интегрирования по заданным параметрам моделирования.

```
enqueueEventWithIntegTimes :: Event () -> Event ()
```

Будучи определенной таким образом, управляемая временем имитационная модель может быть скомбинирована со событийно-ориентированной и процесс-ориентированной моделями. В дистрибутиве Айвики есть соответствующий пример, где моделируется работа камерной печи[12, 17].

[2] На английском языке часто используют близкий термин *activity-oriented*, хотя это может быть не совсем одно и то же.

2.8 Пример: управляемая временем модель

Для иллюстрации управляемой временем парадигмы дискретно-событийного моделирования давайте возьмем нашу старую задачу, которую мы решили в разделе 2.3 с помощью событийно-ориентированной парадигмы и в разделе 2.6 на основе процесс-ориентированной парадигмы. Постановка задачи приводится ниже повторно. Она соответствует документации к SimPy.

> Есть два станка, которые иногда ломаются. Время наработки распределено экспоненциально со средним 1.0, и время восстановления распределено экспоненциально со средним 0.5. Есть двое мастеров. Так что, оба станка могут быть отремонтированы одновременно, если они сломались в то же время. Вывод задает долговременную пропорцию времени наработки. Должны получить значение около 0.66.

Сейчас модель выглядит немного неуклюже. Более того, мы должны масштабировать модельное время. Точки времени, в которые происходят события, более не являются точными.

```
import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
meanRepairTime = 0.5

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

model :: Simulation Results
model =
  do totalUpTime <- newRef 0.0

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

              -- число итераций, в течение которых
              -- станок работает
```

```
upNum <- newRef (-1)

-- число итераций, в течение которых
-- станок сломан
repairNum <- newRef (-1)

-- создать имитационную модель
return $
  do upNum' <- readRef upNum
     repairNum' <- readRef repairNum

     let untilBroken =
           modifyRef upNum $ \a -> a - 1

         untilRepaired =
           modifyRef repairNum $ \a -> a - 1

         broken =
           do writeRef upNum (-1)
              -- станок сломался
              startUpTime' <- readRef startUpTime
              finishUpTime' <- liftDynamics time
              dt' <- liftParameter dt
              modifyRef totalUpTime $
                \a -> a +
                (finishUpTime' - startUpTime')
              repairTime' <-
                liftParameter $
                randomExponential meanRepairTime
              writeRef repairNum $
                round (repairTime' / dt')

         repaired =
           do writeRef repairNum (-1)
              -- станок починили
              t'  <- liftDynamics time
              dt' <- liftParameter dt
              writeRef startUpTime t'
              upTime' <-
                liftParameter $
                randomExponential meanUpTime
              writeRef upNum $
                round (upTime' / dt')

         result | upNum' > 0      = untilBroken
                | upNum' == 0     = broken
```

```
                              | repairNum' > 0  = untilRepaired
                              | repairNum' == 0 = repaired
                              | otherwise       = repaired
                  result

     --  создать два станка с типом Event ()
     m1 <- machine
     m2 <- machine

     -- запустить модели станков
     runEventInStartTime $
       -- в точках интегрирования
       enqueueEventWithIntegTimes $
       do m1
          m2

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

     return $
       results
       [resultSource
        "upTimeProp"
        "The long-run proportion of up time (~ 0.66)"
        upTimeProp]

main =
  printSimulationResultsInStopTime
  printResultSourceInEnglish
  model specs
```

Результаты очень похожи на те, что мы получили ранее, хотя сейчас модель более грубая, и она основана на другом подходе:

```
$ runghc -package=aivika -package=mtl MachRep1TimeDriven.hs
----------

-- simulation time
t = 1000.0

-- The long-run proportion of up time (~ 0.66)
upTimeProp = 0.6556000000000012
```

Тем не менее, управляемая временем парадигма может быть исключительно полезна для моделирования некоторых частей, которые трудны в

реализации на основе других парадигм имитационного моделирования.

Глава 3

Ресурсы

Ресурсы используются, когда нам нужно смоделировать разделяемый, но ограниченный доступ. Ресурс может быть простым, когда дискретный процесс, пытающийся запросить ресурс, приостанавливается в случае нехватки ресурса. Также существует другой тип ресурсов, которые могут быть вытеснены, когда другой процесс с более высоким приоритетом отбирает владение ресурсом у процесса с меньшим приоритетом. Все это поддерживается Айвикой.

3.1 Стратегии очереди

Прежде чем мы перейдем к более высокоуровневым моделирующим конструкциям, нам нужно определить *стратегии очереди*[17], которые будут предписывать то, как одновременные запросы должны быть упорядочены.

В Айвике стратегии очереди выражаются в терминах семейств типов, где каждый экземпляр стратегии очереди может определить свой собственный тип хранилища очереди.

```
class QueueStrategy s where

  data StrategyQueue s :: * -> *

  newStrategyQueue :: s -> Simulation (StrategyQueue s i)
  strategyQueueNull :: StrategyQueue s i -> Event Bool
```

Первая функция создает очередь по заданной стратегии. Вторая проверяет, а пуста ли очередь.

Класс типов `DequeueStrategy` определяет стратегию, которая применяется, когда мы пытаемся удалить из очереди элемент, но при этом есть конкурирующие запросы, которые не могут быть исполнены немедленно. Поэтому мы должны решить, какой же из запросов имеет более высокий приоритет при удалении из очереди.

```
class QueueStrategy s => DequeueStrategy s where
  strategyDequeue :: StrategyQueue s i -> Event i
```

Класс типов `EnqueueStrategy` определяет стратегию, которая применяется, когда мы пытаемся положить элемент в очередь, но при этом есть конкурирующие запросы, которые также не могут быть исполнены немедленно. Мы упорядочиваем запросы некоторым образом, помещая их в очередь запросов.

```
class DequeueStrategy s => EnqueueStrategy s where
  strategyEnqueue :: StrategyQueue s i -> i -> Event ()
```

Также существует версия стратегии добавления, которая использует приоритеты.

```
class DequeueStrategy s => PriorityQueueStrategy s p | s -> p where
  strategyEnqueueWithPriority :: StrategyQueue s i -> p -> i -> Event ()
```

В настоящее время в Айвике определены четыре стандартных стратегии[1]:

- FCFS (First Come - First Served), также известное как FIFO (First In - First Out), когда первым обслуживается первый запрос;
- LCFS (Last Come - First Served), также известное как LIFO (Last In - First Out), когда первым обслуживается последний запрос;
- SIRO (Service in Random Order), когда запросы обслуживаются случайным образом;
- StaticPriorities (Using Static Priorities), когда запросы обслуживаются на основе приоритетов.

Эти стратегии реализованы как типы данных, которые имеют конструктор данных с соответствующим названием.

[1] Вы можете определить свои стратегии.

```
data FCFS = FCFS deriving (Eq, Ord, Show)
data LCFS = LCFS deriving (Eq, Ord, Show)
data SIRO = SIRO deriving (Eq, Ord, Show)
data StaticPriorities = StaticPriorities deriving (Eq, Ord, Show)
```

Каждый тип реализует соответствующую стратегию очереди.

```
instance EnqueueStrategy FCFS
instance EnqueueStrategy LCFS
instance EnqueueStrategy SIRO
instance PriorityQueueStrategy StaticPriorities Double
```

3.2 Ресурс

Ресурс[8] моделирует нечто, к чему возникает очередь.

```
data Resource s
```

Здесь параметр типа `s` представляет собой стратегию очереди. Мы можем использовать как стандартную стратегию, так и задать нашу собственную стратегию очереди. Это будет работать в обоих случаях.

Простейший конструктор позволяет нам создать новый ресурс по заданным стратегии очереди и начальному количеству.

```
newResource :: QueueStrategy s => s -> Int -> Simulation (Resource s)
```

Чтобы запросить заданный ресурс, мы можем использовать следующие функции:

```
requestResource :: EnqueueStrategy s => Resource s -> Process ()

requestResourceWithPriority ::
  PriorityQueueStrategy s p => Resource s -> p -> Process ()
```

Обе приостанавливают дискретный процесс в случае недостатка ресурса до тех пор, пока какая-нибудь другая имитационная активность не освободит ресурс.

```
releaseResourceWithinEvent :: DequeueStrategy s => Resource s -> Event ()
```

Существует также более удобная версия последней функции, которая работает в рамках вычисления `Process`, но приведенная только что функция подчеркивает тот факт, что освобождение ресурса не может заблокировать дискретный процесс, и это действие выполняется немедленно.

```
releaseResource :: DequeueStrategy s => Resource s -> Process ()
```

Мы можем запросить текущее доступное количество заданного ресурса, так же как и запросить его емкость и применяемую стратегию.

```
resourceCount :: Resource s -> Event Int
resourceMaxCount :: Resource s -> Maybe Int
resourceStrategy :: Resource s -> s
```

Вторая функция возвращает опциональное значение, что указывает на то, что максимальное количество может быть и не задано при создании ресурса.

```
newResourceWithMaxCount ::
  QueueStrategy s => s -> Int -> Maybe Int -> Simulation (Resource s)
```

По умолчанию максимальное возможное количество, т.е. емкость ресурса, устанавливается равной начальному количеству, которое задается, если вызвать первый конструктор `newResource`.

Существуют синонимы типов для ресурсов со стандартными стратегиями очереди.

```
type FCFSResource = Resource FCFS
type LCFSResource = Resource LCFS
type SIROResource = Resource SIRO
type PriorityResource = Resource StaticPriorities
```

Есть также конструкторы, которые используют эти синонимы типов. Некоторые из них используются далее в примерах.

```
newFCFSResource :: Int -> Simulation FCFSResource
newLCFSResource :: Int -> Simulation LCFSResource
newSIROResource :: Int -> Simulation SIROResource
newPriorityResource :: Int -> Simulation PiriorityResource
```

Наконец, существует вспомогательная функция `usingResource`, которая запрашивает ресурс, запускает заданное вычисление `Process` и затем освобождает ресурс независимо от того, был этот процесс отменен, или возникла какая исключительная ситуация.

```
usingResource :: EnqueueStrategy s => Resource s -> Process a -> Process a
usingResource r m =
  do requestResource r
     finallyProcess m $ releaseResource r
```

Мы можем сделать то же самое с ресурсом на основе приоритетов.

```
usingResourceWithPriority ::
  PriorityQueueStrategy s p => Resource s -> p -> Process a -> Process a
usingResourceWithPriority r priority m =
  do requestResourceWithPriority r priority
     finallyProcess m $ releaseResource r
```

Обе функции используют функцию `finallyProcess`, которая позволяет запустить завершающую часть в рамках вычисления, чтобы там ни произошло, возникни исключительная ситуация, или если вычисление будет вовсе отменено.

3.3 Пример: использование ресурсов

Чтобы показать то, как ресурсы могут быть использованы для моделирования, давайте снова возьмем задачу из документации к SimPy[15], только другую.

> Есть два станка, которые иногда ломаются. Время наработки распределено экспоненциально со средним 1.0, а время восстановления тоже распределено экспоненциально, но со средним 0.5 В этом примере есть только один мастер, так что оба станка не могут быть восстановлены одновременно, если сломаются в то же самое время.
>
> Помимо долговременной пропорции времени наработки, давайте также найдем долговременную пропорцию того времени, когда в момент поломки станка мастер был свободен и не был занят починкой другого станка. На выходе должны получить значения примерно 0.6 и 0.67.

В Айвике эта задача может быть решена следующим образом.

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

```
import Simulation.Aivika

meanUpTime = 1.0
meanRepairTime = 0.5

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

model :: Simulation Results
model =
  do -- число поломок станков
     nRep <- newRef 0

     -- число поломок, когда мастер приступил
     -- к починке немедленно
     nImmedRep <- newRef 0

     -- общее время наработки станков
     totalUpTime <- newRef 0.0

     repairPerson <- newFCFSResource 1

     let machine :: Process ()
         machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)

              -- проверить доступность мастера
              liftEvent $
                do modifyRef nRep (+ 1)
                   n <- resourceCount repairPerson
                   when (n == 1) $
                     modifyRef nImmedRep (+ 1)

              requestResource repairPerson
              repairTime <-
                randomExponentialProcess meanRepairTime
              releaseResource repairPerson

              machine
```

```
     runProcessInStartTime machine
     runProcessInStartTime machine

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

         immedProp :: Event Double
         immedProp =
           do n <- readRef nRep
              nImmed <- readRef nImmedRep
              return $
                fromIntegral nImmed /
                fromIntegral n

     return $
       results
       [resultSource
        "upTimeProp"
        "Долговременная пропорция времени наработки (~ 0.6)"
        upTimeProp,
        --
        resultSource
        "immedProp"
        "Долговременная пропорция немедленного доступа (~ 0.67)"
        immedProp]

main =
  printSimulationResultsInStopTime
  printResultSourceInRussian
  model specs
```

Это законченная программа на языке Haskell, которая может быть запущена с помощью утилиты runghc или скомпилирована в нативный код. Она возвращает ожидаемые результаты.

```
$ runghc -package=aivika -package=mtl MachRep2.hs
----------

-- модельное время
t = 1000.0

-- Долговременная пропорция времени наработки (~ 0.6)
upTimeProp = 0.6089071789353254
```

```
-- Долговременная пропорция немедленного доступа (~ 0.67)
immedProp = 0.6650082918739635
```

Здесь мы использовали дополнительную сущность в виде ресурса. На самом деле, в Айвике есть разные ресурсы. Тот ресурс, что мы применили, является самым простым, который оптимизирован для скорости выполнения.

3.4 Статистика по ресурсу

Существует другой вид ресурсов, который собирает статистику во время имитации. Такой ресурс собирает данные о размере очереди, времени ожидания и коэффициенте использования ресурса.

Во-первых, соответствующий тип находится в другом модуле, из-за чего нам следует использовать квалифицированный импорт подобный этому

```
import qualified Simulation.Aivika.Resource as R
```

Поскольку некоторые из этих данных зависят от начального времени, то мы создаем ресурс уже внутри вычисления `Event`, используя функцию запуска `runEventInStartTime` тогда, когда необходимо.

```
newResource :: QueueStrategy s => s -> Int -> Event (Resource s)

newResourceWithMaxCount ::
  QueueStrategy s => s -> Int -> Maybe Int -> Event (Resource s)
```

В течение имитации мы всегда можем получить текущее состояние статистики с помощью функций, подобной этой

```
resourceWaitTime :: Resource s -> Event (SamplingStats Double)
```

Она возвращает статистику по времени ожидания ресурса. Здесь тип данных `SamplingStats` соответствует статистике на основе наблюдений, которая наряду с другим видом статистики описана в главе 4 более подробно.

Однако в большинстве случаев нет необходимости запрашивать явно статистику, используя такие функции. Вместо этого мы будем запрашивать статистику на уровне обработки результатов моделирования, как будет показано в примере из раздела 3.7.

Наконец, мы можем сбросить статистику по ресурсу, вызвав следующую функцию.

```
resetResource :: Resource s -> Event ()
```

Нам обычно следует вызвать ее в некоторой точке модельного времени, используя функцию `enqueueEvent`, чтобы активировать соответствующее событие для удаления эффектов не совсем чистого старта, когда модель могла еще не находиться в стабильном состоянии.

В остальном этот ресурс имеет почти те же самые функции, что мы рассмотрели в предыдущем разделе 3.2.

3.5 Пример: сбор статистики по ресурсу

Давайте возьмем снова нашу модель из раздела 3.3, но только перепишем ее, используя ресурс, который собирает статистику во время имитации.

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

import Simulation.Aivika
import qualified Simulation.Aivika.Resource as R

meanUpTime = 1.0
meanRepairTime = 0.5

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

model :: Simulation Results
model =
  do -- число поломок станков
     nRep <- newRef 0

     -- число поломок, когда мастер приступил
     -- к починке немедленно
     nImmedRep <- newRef 0

     -- общее время наработки станков
     totalUpTime <- newRef 0.0

     repairPerson <- runEventInStartTime $
                     R.newFCFSResource 1
```

```
let machine :: Process ()
    machine =
      do upTime <-
           randomExponentialProcess meanUpTime
         liftEvent $
           modifyRef totalUpTime (+ upTime)

         -- проверить доступность мастера
         liftEvent $
           do modifyRef nRep (+ 1)
              n <- R.resourceCount repairPerson
              when (n == 1) $
                modifyRef nImmedRep (+ 1)

         R.requestResource repairPerson
         repairTime <-
           randomExponentialProcess meanRepairTime
         R.releaseResource repairPerson

         machine

runProcessInStartTime machine
runProcessInStartTime machine

let upTimeProp =
      do x <- readRef totalUpTime
         y <- liftDynamics time
         return $ x / (2 * y)

    immedProp :: Event Double
    immedProp =
      do n <- readRef nRep
         nImmed <- readRef nImmedRep
         return $
           fromIntegral nImmed /
           fromIntegral n

return $
  results
  [resultSource
   "upTimeProp"
   "Долговременная пропорция времени наработки (~ 0.6)"
   upTimeProp,
   --
   resultSource
```

```
        "immedProp"
        "Долговременная пропорция немедленного доступа (~ 0.67)"
        immedProp,
        --
        resultSource
        "repairPerson"
        "Мастер по ремонту"
        repairPerson]

main =
  printSimulationResultsInStopTime
  printResultSourceInRussian
  model specs
```

Это почти та же самая модель, что и предыдущая. Только мы импортируем ресурс, используя квалифицированный импорт с буквой `R`.

Теперь если мы попытаемся запустить имитацию, то мы увидим нечто новое: детальный и многословный вывод о состоянии статистики по ресурсу:

```
$ runghc -package=aivika -package=mtl MachRep2.hs
----------

-- модельное время
t = 1000.0

-- Долговременная пропорция времени наработки (~ 0.6)
upTimeProp = 0.5996313437932468

-- Долговременная пропорция немедленного доступа (~ 0.67)
immedProp = 0.6593137254901961

-- Мастер по ремонту
repairPerson:

  -- текущая длина очереди к ресурсу
  queueCount = 0

  -- статистика длины очереди к ресурсу
  queueCountStats = { count = 835, mean = 0.19555148877102507,
    std = 0.396624638683051, min = 0 (t = 0.0),
    max = 1 (t = 1.029005479507997), t in [0.0, 992.0556725133001] }

  -- общее время ожидания ресурса
  totalWaitTime = 193.99796370371632
```

```
-- время ожидания ресурса
waitTime = { count = 1224, mean = 0.15849506838538902,
  std = 0.3429091438416053, min = 0.0, max = 2.683002286836768 }

-- текущее доступное количество ресурса
count = 1

-- статистика по доступному количеству ресурса
countStats = { count = 1615, mean = 0.394321376208336,
  std = 0.48870443876999925, min = 0 (t = 0.7751655037553654),
  max = 1 (t = 0.0), t in [0.0, 999.8392739834163] }

-- текущее используемое количество ресурса
utilisationCount = 0

-- статистика по используемому количеству ресурса
utilisationCountStats = { count = 2449, mean = 0.6056786237916639,
  std = 0.48870443876999925, min = 0 (t = 0.0),
  max = 1 (t = 0.7751655037553654), t in [0.0, 999.8392739834163] }
```

Вывод довольно таки хорошо говорит сам за себя. Он показывает длину очереди, ее статистику, время ожидания и соответствующую статистику, а также похожую информацию по количеству ресурса и тому, как много использовалось ресурса. В главе 4 вы найдете больше подробностей о типах, которые используются для сбора статистики в Айвике. Здесь мы видим их строковое представление.

Ранее мы выводили разные графики, которые рисовались во время имитационных экспериментов. Как мы увидим далее, мы можем также использовать свойства ресурсов при запросе данных во время рисования графиков. Более того, посыл в том, что мы можем запрашивать такие данные по произвольным свойствам объектов, которые поддерживают протокол типа `Results`, то есть, тех объектов, которые могут быть возвращены в результатах моделирования, например, очередей и серверов, если перечислить некоторые.

3.6 Обращение к свойствам

Как мы увидели, ресурсы могут иметь дополнительные поля с данными статистики. Мы можем ссылаться на эти данные, либо вызывая прямые функции доступа к данным, либо задавая, какие именно результаты мы хотим использовать. Здесь мы рассмотрим второй подход.

Итак, существует следующий модуль, который содержит вспомогательные функции для работы с результатами моделирования.

```
module Simulation.Aivika.Results.Transform
```

Ранее мы видели, что мы можем задать, какой временной ряд мы хотим использовать во время рисования графика. Точно таким же образом мы можем задать, что мы хотим использовать временной ряд, соответствующий длине очереди ресурса:

```
resourceCount :: Resource -> ResultTransform
```

`ResultTransform` представляет собою результат моделирования, в данном случае это будет временной ряд, но тип `Resource` — это не из тех типов ресурсов, что мы видели прежде. В рамках рассматриваемого модуля, это отдельный тип, который имеет конструктор, позволяющий создать новый ресурс этого вида по заданному результату моделирования.

```
newtype Resource = Resource ResultTransform
```

Как вы можете помнить, мы ссылаемся на временные ряды по их названиям. Так, мы ссылаемся на некоторый ресурс по его названию, а затем вызываем конструктор данных `Resource`, чтобы создать новое значение типа данных `Resource`. Затем мы можем получить доступ к дополнительным временным рядам с помощью функций подобных `resourceCount`. Ресурс сам по себе является составным объектом, который не может быть нарисован непосредственно на графике, так как он состоит из множества временных рядов.

Мы можем написать что-то похожее на это

```
r = Resource $ resultByName "repairPerson"
s = resourceCount r
```

Ниже мы выведем график для используемого количества ресурса. Для этого нам понадобится следующая функция, которая возвращает соответствующий временной ряд.

```
resourceUtilisationCount :: Resource -> ResultTransform
```

3.7 Пример: график для свойства ресурса

Сейчас мы увидим, как мы можем вывести график для используемого количества ресурса из нашего предыдущего примера. Как и прежде, мы разобьем код на разные части, где первый файл задаст саму модель, второй файл опишет эксперимент, а третий файл запустит имитацию, используя один из интерфейсов графиков.

3.7.1 Возвращение результатов из модели

Сейчас мы слегка перепишем код, поместив имитационную модель в отдельный модуль. Нет никакой существенной разницы с кодом, приведенным в разделе 3.5.

```
module Model (model) where

import Control.Monad
import Control.Monad.Trans

import Simulation.Aivika
import qualified Simulation.Aivika.Resource as R

meanUpTime = 1.0
meanRepairTime = 0.5

model :: Simulation Results
model =
  do -- число поломок станков
     nRep <- newRef 0

     -- число поломок, когда мастер приступил
     -- к починке немедленно
     nImmedRep <- newRef 0

     -- общее время наработки станков
     totalUpTime <- newRef 0.0

     repairPerson <- runEventInStartTime $
                     R.newFCFSResource 1

     let machine :: Process ()
         machine =
           do upTime <-
```

```
            randomExponentialProcess meanUpTime
          liftEvent $
            modifyRef totalUpTime (+ upTime)

          -- проверить доступность мастера
          liftEvent $
            do modifyRef nRep (+ 1)
               n <- R.resourceCount repairPerson
               when (n == 1) $
                 modifyRef nImmedRep (+ 1)

          R.requestResource repairPerson
          repairTime <-
            randomExponentialProcess meanRepairTime
          R.releaseResource repairPerson

          machine

  runProcessInStartTime machine
  runProcessInStartTime machine

  let upTimeProp =
        do x <- readRef totalUpTime
           y <- liftDynamics time
           return $ x / (2 * y)

      immedProp :: Event Double
      immedProp =
        do n <- readRef nRep
           nImmed <- readRef nImmedRep
           return $
             fromIntegral nImmed /
             fromIntegral n

  return $
    results
    [resultSource
     "upTimeProp"
     "The long-run proportion of up time (~ 0.6)"
     upTimeProp,
     --
     resultSource
     "immedProp"
     "The proption of time of immediate access (~0.67)"
     immedProp,
     --
```

```
        resultSource
        "repairPerson"
        "The repair person"
        repairPerson]
```

3.7.2 Задание эксперимента

Сейчас последует наиболее удивительная вещь. Мы определяем эксперимент декларативно. Мы хотим вывести тренд для используемого количества ресурса. График также покажет доверительные интервалы по правилу 3-х сигм.

```
module Experiment (experiment, generators) where

import Data.Monoid

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

import qualified Simulation.Aivika.Results.Transform as T

specs = Specs { spcStartTime = 0,
                spcStopTime = 1000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1000,
    experimentTitle = "MachRep2",
    experimentDescription = "Example: Charts for Resource Properties" }

upTimeProp = resultByName "upTimeProp"
immedProp = resultByName "immedProp"

repairPerson = T.Resource $ resultByName "repairPerson"
repairPersonUtil = T.resourceUtilisationCount repairPerson

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView $ defaultDeviationChartView {
```

```
    deviationChartTitle = "Resource Utilization",
    deviationChartLeftYSeries = repairPersonUtil },
  outputView $ defaultFinalStatsView {
    finalStatsTitle = "Resource Utilization Stats.",
    finalStatsSeries = repairPersonUtil }]
```

Мы также задаем, что итоговая веб-страница покажет нам сводную статистику по используемому количеству ресурса. Число имитационных запусков равно 1000.

Обратите внимание на то, как мы получаем доступ к свойствам ресурса. Мы трактуем заданный источник результата как ресурс, а затем безопасно вызываем соответствующее свойство.

3.7.3 Вывод графиков

Теперь мы выбираем один из интерфейсов графиков. Код точно такой же, каким он был в разделе 1.4.3.

3.7.4 Запуск имитационного эксперимента

Имитационный эксперимент, состоящий из 1000 запусков, скомпилированный с опцией `-O2` и запущенный с опциями `+RTS -N`, длился около 3 секунд на моем компьютере, имеющем двухъядерный процессор с гипертредингом, то есть, с четырьмя виртуальными ядрами. Если у вас 8-ядерный или 32-ядерный процессор, то скорость имитации будет еще более быстрой.

Таблица 3.1: Статистика использования мастера по ремонту, собранная по данным в конечных точках времени.

mean	0.5950000000000005
deviation	0.4911376754192414
minimum	0.0
maximum	1.0
count	1000

Вы можете найти соответствующий график на рисунке 3.1, который пока-

зывает тренд и доверительные интервалы по правилу 3-х сигм.

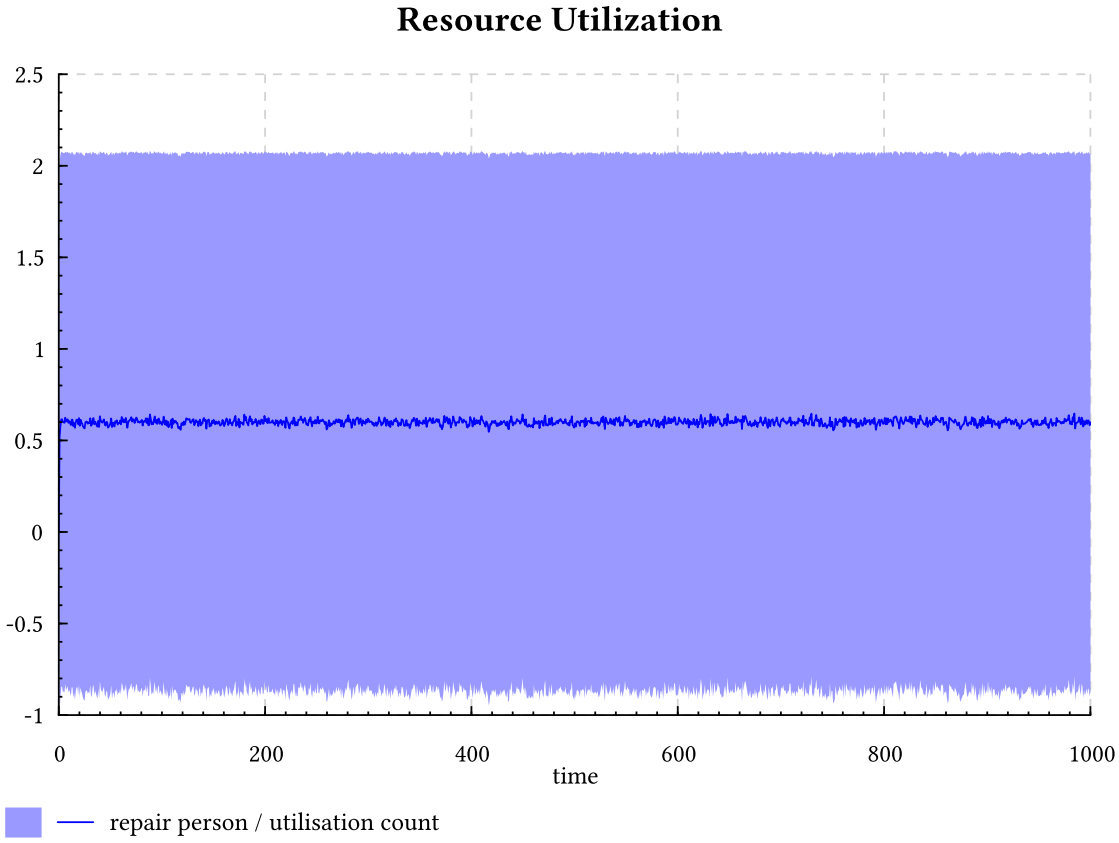


Рис. 3.1: График отклонения для используемого количества ресурса.

3.8 Вытеснение ресурса

Вычисление `Process` поддерживает особое и очень полезное свойство, которое называется *вытеснением*. Есть быть не очень формально строгим, то вытеснение подразумевает, что некоторый дискретный процесс может быть временно прерван извне таким образом, что он может потом возобновить свое выполнение. Это влияет на статистику, влияет на владение ресурсами и т.п.

В Айвике вытеснение процесса скрыто от непосредственного использования. Вместо этого существуют некоторые типы данных, которые предоставля-

ют высокоуровневый интерфейс, который уже использует вытеснение внутри своей реализации.

Одним из таких типов данных является особый вид ресурсов, рассматриваемых в этом разделе. Как и прежде, существует два похожих модуля, которые определяют тип ресурса: один — более простой, но оптимизированный для исполнения, а другой является более полной версией ресурса, который обновляет свою статистику. Для простоты рассмотрим последний.

Этот ресурс располагается в своем собственном модуле и имеет предопределенную стратегию очереди. Так, нет необходимости задавать стратегию, как мы делали для более простых ресурсов в предыдущих разделах.

```
module Simulation.Aivika.Resource.Preemption

data Resource
```

Как и ранее, существует две конструирующих функции, которые позволяют создать новый ресурс. Есть обязательное начальное количество и опциональная емкость:

```
newResource :: Int -> Event Resource
newResourceWithMaxCount :: Int -> Maybe Int -> Event Resource
```

Чтобы захватить ресурс, дискретный процесс должен предоставить свой приоритет. Если ресурс уже принадлежит другому процессу с меньшим приоритетом, то тогда старый процесс вытесняется, и текущий процесс получает ресурс во владение. Иначе, если старый процесс имеет такой же приоритет или выше, то тогда текущий процесс ждет освобождения ресурса. Обратите внимание, что меньшее значение означает более высокий приоритет.

```
requestResourceWithPriority :: Resource -> Double -> Process ()
```

Для освобождения ресурса мы вызываем следующую функцию. Если некоторый процесс был вытеснен до этого, то тогда тот процесс возобновит свое выполнение с того места, где он был вытеснен.

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

К сожалению, текущая реализация вытеснения ресурса имеет ограничение. Ресурс может быть освобожден только тем процессом, который захватил ресурс прежде. Если вам нужно более сложное и гибкое поведение, то, вероятно, вам следует рассмотреть возможность использования GPSS-подобного предметно-ориентированного языка, описанного в главе 10.

Более того, освобождение данного типа вытесняемого ресурса имеет сложность, пропорциональную количеству процессов, владеющих ресурсом. Поэтому старайтесь избегать случаев с большим количеством владельцев ресурса! К слову сказать, в случае GPSS у аналогичного по свойствам так называемого прибора может быть всего один владелец. Поэтому там освобождение прибора является достаточно быстрой операцией.

Для сброса статистики по ресурсу в некоторой точке модельного времени мы можем вызвать следующую функцию.

```
resetResource :: Resource -> Event ()
```

В разделе 6.8 вы можете найти пример использования вытеснения ресурса, но для этого нам нужно ввести некоторые новые концепции, которые рассмотрены далее.

Глава 4

Статистика

Сбор и обработка статистики — важная часть имитации. Айвика использует подход, где сводная статистика трактуется как неизменяемая структура данных, что упрощает программирование и делает имитацию более надежной.

Существует два разных типа статистики, которую мы можем собирать. Первый тип основан на наблюдениях, тогда как второй основан на выборке, зависящей от модельного времени.

4.1 Статистика на основе наблюдений

Тип данных `SamplingStats` используется для аккумулирования статистики, основанной на наблюдениях. Примером является время ожидания в очереди.

```
data SamplingStats a

class Num a => SamplingData a where

  emptySamplingStats :: SamplingStats a
  addSamplingStats :: a -> SamplingStats a -> SamplingStats a
  combineSamplingStats :: SamplingStats a
                          -> SamplingStats a
                          -> SamplingStats a
```

Первая функция возвращает пустую статистику, которая говорит о том, что не было данных. Функция `addSamplingStats` принимает новое значение выборки и некоторую статистику, а затем возвращает новую аккумулированную

статистику. Третья функция принимает две статистики и возвращает комбинированную статистику. Другими словами, мы имеем моноид с единичным элементом.

Существует две важных реализации класса типов `SamplingData`, которые позволяют собирать числовую статистику.

```
instance SamplingData Int
instance SamplingData Double
```

Обычной ошибкой является то, когда пытаются использовать довольно тяжеловесный тип `Var` для сбора статистики. Тем не менее, строго рекомендуется использовать ссылку `Ref` для итеративного обновления легковесных значений `SamplingStats`, что является более эффективным и более простым методом.

По заданному значению `SamplingStats` мы можем вывести сводную статистику, которая бы включала в себя число наблюдений, минимальное, максимальное, среднее значения, среднее квадратичное значение, дисперсию и средне-квадратичное отклонение, соответственно.

```
samplingStatsCount :: SamplingStats a -> Int
samplingStatsMin   :: SamplingStats a -> a
samplingStatsMax   :: SamplingStats a -> a
samplingStatsMean  :: SamplingStats a -> Double
samplingStatsMean2 :: SamplingStats a -> Double
samplingStatsVariance :: SamplingStats a -> Double
samplingStatsDeviation :: SamplingStats a -> Double
```

Наконец, статистика `SamplingStats` может быть возвращена как источник результата `ResultSource` из имитационной модели.

4.2 Статистика с привязкой ко времени

Статистика `TimingStats` очень похожа на `SamplingStats`, но только первая позволяет использовать выборку, привязанную к точкам модельного времени. Соответствующую случайную величину на английском языке часто называют *time-persistent*[12]. Примером такой случайной величины является длина очереди.

На русском языке для обозначения самой статистики используется иногда термин *непрерывной статистики*[16]. Для краткости мы будет использовать термин *временной статистики*, поскольку он ближе к названию соответствующего типа данных в Айвике.

```
data TimingStats a

class Num a => TimingData a where

  emptyTimingStats :: TimingStats a
  addTimingStats :: Double -> a -> TimingStats a -> TimingStats a
```

Как и прежде, мы можем создать пустую статистику, но во время обновления статистики, мы должны также задать соответствующее время первым аргументом, передав его функции `addTimingStats`. Более того, нет аналога комбинирующей функции для этого типа статистики. Поэтому это не моноид.

Между прочим, именно из-за необходимости задавать время конструктор ресурса из раздела 3.4 возвращает действие внутри вычисления `Event`, а не внутри вычисления `Simulation`, как мы могли бы ожидать. Мы просто инициализируем некоторую статистику по ресурсу непосредственно в момент создания ресурса.

`TimingStats` возвращает больше данных. Во-первых, эта статистика возвращает то же самое множество свойств: число наблюдений, минимально, максимальное, среднее значения, среднее квадратичное значение, дисперсию и средне-квадратичное отклонение.

```
timingStatsCount :: TimingStats a -> Int
timingStatsMin   :: TimingStats a -> a
timingStatsMax   :: TimingStats a -> a
timingStatsMean  :: TimingData a => TimingStats a -> Double
timingStatsMean2 :: TimingData a => TimingStats a -> Double
timingStatsVariance :: TimingData a => TimingStats a -> Double
timingStatsDeviation :: TimingData a => TimingStats a -> Double
```

Здесь число наблюдений уже имеет довольно условное значение в отличие от предыдущего типа статистики, основанной на наблюдениях.

В дополнение, значение `TimingStats` возвращает последнее аккумулированное значение, модельное время, в которое был зафиксирован минимум, также время, когда был зафиксирован максимум, время начала выборки, последнее время выборки, сумма значений и сумма квадратов значений, соответственно.

```
timingStatsLast :: TimingStats a -> a
timingStatsMinTime :: TimingStats a -> Double
timingStatsMaxTime :: TimingStats a -> Double
timingStatsStartTime :: TimingStats a -> Double
timingStatsLastTime :: TimingStats a -> Double
timingStatsSum :: TimingStats a -> Double
timingStatsSum2 :: TimingStats a -> Double
```

Существуют так же две наиболее важных реализации класса `TimingData`, которые позволяют собирать числовую статистику данного типа.

```
instance TimingData Int
instance TimingData Double
```

Как и прежде, значение `TimingStats` может быть возвращено из модели как источник результата `ResultSource`.

Мы можем видеть, что только два первых момента вычисляются этими двумя типами `SamplingStats` и `TimingStats`, что должно покрывать большинство вариантов использования. Если вам нужен более детальный статистический анализ, то, вероятно, вам нужно будет собирать и обрабатывать статистику самим.

Глава 5

Сигналы и задачи

В Айвике есть сигнальный механизм, который основан на идее интерфейса `IObservable` из .NET. Сигнал представляет собой нечто, что уведомляет своих слушателей об изменениях, событиях и т.п. Слушатель может подписаться на получение значений сигнала, а затем отписаться.

Мы также можем определить задачу для представления дискретного процесса, запущенного в фоне. Испустив соответствующий сигнал, задача уведомит нас о том, что процесс завершился и вернул результат.

5.1 Сигналы

Следующий моноид представляет собой сигнал, который уведомляет о том, что произошло некоторое событие.

```
data Signal a =
  Signal { handleSignal :: (a -> Event ()) -> Event DisposableEvent }
```

Функция `handleSignal` принимает некоторый сигнал и его обработчик, подписывает обработчик на получение значений сигнала, а затем возвращает вложенное вычисление типа `DisposableEvent`, которое, будучи примененным, отпишет соответствующий обработчик от получения сигнала.

Действие, в течение которого мы отписываемся от сигнала, происходит в модельное время. Поэтому возвращенное вложенное вычисление на самом деле имеет тип `Event ()`, скрытый за удобным названием типа.

```
disposeEvent :: DisposableEvent -> Event ()
```

Если мы вовсе не собираемся отписываться, то тогда мы можем проигнорировать вложенное вычисление.

```
handleSignal_ :: Signal a -> (a -> Event ()) -> Event ()
```

Мы можем трактовать сигналы в функциональном стиле, преобразуя их, соединяя или фильтруя с помощью комбинаторов.

```
instance Monoid (Signal a)
instance Functor Signal

filterSignal :: (a -> Bool) -> Signal a -> Signal a
```

Ссылка `Ref` и переменная `Var` могут уведомлять об изменении своего состояния, предоставляя соответствующие сигналы.

```
refChanged :: Ref a -> Signal a
varChanged :: Var a -> Signal a
```

Мы можем создать источник сигнала вручную. Различая источник от сигнала, мы можем публиковать сигнал с помощью чистой функции. Тем не менее, мы должны испускать сигнал в рамках вычисления, синхронизированного с очередью событий.

```
data SignalSource a

newSignalSource :: Simulation (SignalSource a)

publishSignal :: SignalSource a -> Signal a
triggerSignal :: SignalSource a -> a -> Event ()
```

5.2 Задачи

Существует связь между сигналами и дискретными процессами, которая выражается следующей функцией, что приостанавливает текущий процесс, пока не придет сигнал.

```
processAwait :: Signal a -> Process a
```

Ожидается только одно значение сигнала, а затем процесс автоматически отпишется от заданного сигнала.

В Айвике есть обратное преобразование из вычисления `Process` в значение `Signal`, но оно немного усложнено из-за того, что процесс может быть, на самом деле, отменен, или исключение `IO` может произойти во время имитации. Соответствующее преобразование определено с помощью типа `Task`.

```
runTask :: Process a -> Event (Task a)
```

Здесь мы запускаем в фоне заданный процесс и немедленно возвращаем соответствующую задачу внутри вычисления `Event`. Позже мы можем запросить результат запущенного вычисления `Process`, чтобы узнать, завершилось ли оно успешно, возникла ли ошибка `IO`, или было ли отменено вычисление? Пожалуйста, обратитесь к документации Айвики за подробностями.

С использованием сигналов реализована функция `timeoutProcess`, которая упоминалась ранее. Она позволяет нам запустить подпроцесс в пределах заданного тайм-аута. Функция создает внутренний источник сигнала. Запущенный подпроцесс пытается вычислить результат и в случае успеха уведомить родительский процесс, испустив соответствующий сигнал. Только еще раз стоит заметить, что функция `timeoutProcess` довольно-таки медленная.

Сигналы также используются в функции `memoProcess`. Эта функция принимает заданный процесс и возвращает новый результирующий процесс, который всегда возвращает то же самое значение в течение текущего имитационного запуска независимо от того, сколько раз тот процесс будет запущен. Значение вычисляется только один раз для каждого имитационного запуска, а другие экземпляры внешних процессов просто ожидают сигнала для получения результата.

Наконец, сигналы широко используются в имитационных экспериментах. Именно по испущенному соответствующему сигналу инструментальные средства вывода графиков и других результатов моделирования получают информацию о том, что заданный источник результата изменился.

5.3 Композиты

Вы могли заметить, что действие, в течение которого мы подписываемся на получение значений сигнала, возвращает значение `DisposableEvent` в рамках вычисления `Event`. Это настолько общий шаблон, что была введена отдельная монада.

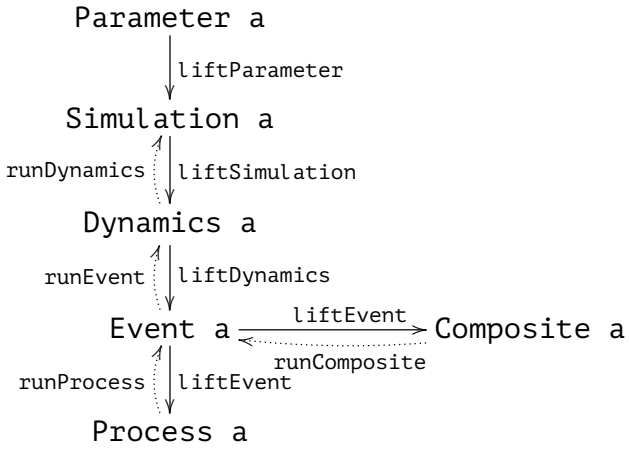
```
data Composite a

runComposite :: Composite a
                -> DisposableEvent
                -> Event (a, DisposableEvent)

runComposite_ :: Composite a -> Event a
```

Первая из этих функций позволяет создать заданный композит, который может быть затем разобран сразу после вызова соответствующего вычисления `DisposableEvent`.

При создании композита мы можем добавить наши собственные будущие действия, которые будут применены во время разбора композита. Это главная особенность рассматриваемого вычисления.

```
disposableComposite :: DisposableEvent -> Composite ()
```

Вычисление `Composite` основано на вычислении `Event`. По сути, это такое вычисление `Event`, которое запоминает все свои действия `DisposableEvent`, что могут быть затем применены для откатов таких действий при разборе композита. Вычисление `Composite` используется часто вместе с сигналами.



Глава 6

Сети очередей

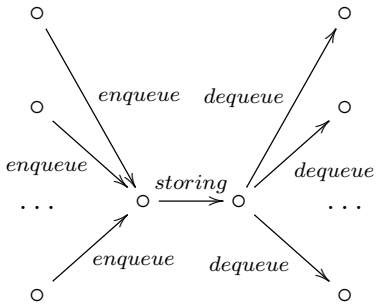
6.1 Очереди

Иногда нам нужно выделить в сети некоторое место, где моделируемые сущности ожидали бы своей обработки[12]. В Айвике это моделируется с помощью конечных и бесконечных очередей.

Для краткости только конечные очереди рассмотрены ниже, так как это более сложный случай.

```
data Queue si sm so a
```

Этот тип представляет собой очередь с заданными стратегиями для добавления (input), `si`, сохранения внутри (in memory), `sm`, и извлечения (output), `so`, где параметр типа `a` обозначает тип элементов, хранимых в очереди.



Существуют синонимы типов для наиболее важных вариантов использования:

```
type FCFSQueue a = Queue FCFS FCFS FCFS a
type LCFSQueue a = Queue FCFS LCFS FCFS a
type SIROQueue a = Queue FCFS SIRO FCFS a
type PriorityQueue a = Queue FCFS StaticPriorities FCFS a
```

Эти синонимы используют стратегию FIFO как для запросов на добавление, так и на извлечение, что кажется разумным для большинства случаев. Тем не менее, мы можем определить конечную очередь, которая бы обрабатывала все запросы на извлечение, например, случайным образом (SIRO) или в обратном порядке (LIFO) в случае добавления элемента в пустую очередь. Здесь мы оперируем теми же стратегиями очереди, которые мы ввели в разделе 3.1.

Существуют разные функции, которые позволяют создать новую очередь с заданной емкостью.

```
newQueue :: (QueueStrategy si, QueueStrategy sm, QueueStrategy so)
            => si -> sm -> so -> Int -> Event (Queue si sm so a)

newFCFSQueue :: Int -> Event (FCFSQueue a)
newLCFSQueue :: Int -> Event (LCFSQueue a)
newSIROQueue :: Int -> Event (SIROQueue a)
newPriorityQueue :: Int -> Event (PriorityQueue a)
```

Очередь создается внутри вычисления `Event`, так как нам нужно знать текущее модельное время, чтобы начать собирать временную статистику для длины очереди. Статистика инициализируется во время вызова вычисления.

Существуют разные функции для добавления в очередь. Наиболее простая приведена ниже.

```
enqueue :: (EnqueueStrategy si, EnqueueStrategy sm, DequeueStrategy so)
           => Queue si sm so a -> a -> Process ()
```

Она приостанавливает процесс, если конечная очередь заполнена. Поэтому это действие возвращается как вычисление `Process`.

Мы также можем попытаться добавить заданный элемент, и если очередь заполнена, то тогда элемент считается потерянным.

```
enqueueOrLost :: (EnqueueStrategy sm, DequeueStrategy so)
                 => Queue si sm so a -> a -> Event Bool
```

Это действие уже не может приостановить имитационную активность, но оно возвращает вычисление `Event` для флага, указывающего на то, был ли успешно добавлен элемент в очередь.

Простейшая операция извлечения из очереди приостанавливает процесс на то время, пока очередь пуста. Результатом снова является вычисление `Process`.

```
dequeue :: (DequeueStrategy si, DequeueStrategy sm, EnqueueStrategy so)
           => Queue si sm so a -> Process a
```

Здесь сами сигнатуры типов показывают, может ли соответствующее действие приостановить имитационную активность, или, что действие выполняется немедленно.

Есть похожие функции для добавления и извлечения из очереди, которые позволяют задать приоритеты при условии, что соответствующая очередь поддерживает их.

Очередь имеет множество счетчиков, которые обновляются во время имитации. В действительности, эти счетчики являются тем, что нас более всего интересует.

Например, мы можем запросить статистику по длине очереди и времени ожидания. Мы рассмотрели типы данных статистики в главе 4.

```
queueCountStats :: Queue si sm so a -> Event (TimingStats Int)
queueWaitTime :: Queue si sm so a -> Event (SamplingStats Double)
```

Здесь `SamplingStats` — относительно легковесный и неизменяемый тип данных, который хранит в себе сводную статистику, собранную для времени ожидания в очереди. Он возвращает среднее значение, дисперсию, отклонение, минимальное и максимальное значения, а также число наблюдений. `TimingStats` предоставляет дополнительную информацию о модельном времени, когда минимальное и максимальное значения были достигнуты. Также временная статистика принимает во внимание модельное время, в которое данные собираются.

Чтобы сбросить статистику по очереди в текущем модельном времени, мы можем вызвать следующую функцию:

```
resetQueue :: Queue si sm so a -> Event ()
```

Мы можем запросить свойства очереди в имитационном эксперименте подобно тому, как мы запрашивали свойства ресурса в разделе 3.6.

```
import qualified Simulation.Aivika.Results.Transform as T

q = T.Queue $ resultByName "someQueue"
k = T.tr $ T.queueWaitTime q
```

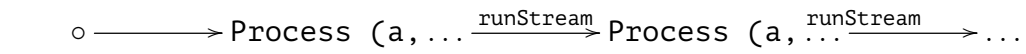
Ниже показано, как очереди могут быть обработаны, используя более высокоуровневые вычисления, которые оперируют потоками данных.

6.2 Поток

Многие вещи значительно упростятся для рассуждений и понимания после того, как мы введем понятие *бесконечного потока* данных, которые приходят с некоторой задержкой модельного времени.

```
newtype Stream a = Cons { runStream :: Process (a, Stream a) }
```

Это разновидность знаменитой *ячейки cons*, где ячейка только возвращается уже внутри вычисления `Process`. Это означает, что поток данных может быть распределен в модельном времени, и что могут быть промежутки между приходом последовательных данных.



Поток представляет собой последовательные данные. Однако, если вам нужно параллельно обрабатывать транзакты, то тогда, вероятно, вам следует посмотреть на GPSS-подобный предметно-ориентированный язык, который также поддерживается Айвикой. Он описан далее в главе 10. Этот предметно-ориентированный язык появился в Айвике позже, и вероятно, что вам следует начать именно с него. При этом, потокам там тоже нашлось место (для задания генераторов транзактов) наряду с сигналами.

Потоки сами по себе хорошо известны в функциональном программировании в течение долгого времени[1]. Очевидно, что их можно фильтровать и преобразовывать.

Сейчас более интересно то, какие новые свойства мы можем получить, введя вычисление `Process` в определение ячейки cons. По крайней мере, тип `Stream` является моноидом.

Усыпляя навсегда нижележащий процесс[1], мы получаем поток, который никогда не возвращает данных.

```
emptyStream :: Stream a
```

Более того, мы можем слить два потока, применяя стратегию `FCFS` при добавлении входных данных.

```
mergeStreams :: Stream a -> Stream a -> Stream a
```

[1]Тем не менее, нижележащий процесс все же может быть отменен.

На самом деле, последная функция является частным случаем более общей функции, которая позволяет соединять потоки в один подобно *мультиплексору*.

```
concatStreams :: [Stream a] -> Stream a
concatStreams = concatQueuedStreams FCFS

concatQueuedStreams :: EnqueueStrategy s => s -> [Stream a] -> Stream a

concatPriorityStreams ::
  PriorityQueueStrategy s p => s -> [Stream (p, a)] -> Stream a
```

Функции используют ресурсы для соединения разных бесконечных потоков данных.

Существует обратная возможность разбить входной поток на заданное число выходных потоков подобно *демультиплексору*. Мы должны проделывать такое, чтобы моделировать параллельную работу по обслуживанию.

```
splitStream :: Int -> Stream a -> Simulation [Stream a]
splitStream = splitStreamQueueing FCFS

splitStreamQueueing ::
  EnqueueStrategy s
  => s -> Int -> Stream a -> Simulation [Stream a]

splitStreamPrioritising ::
  PriorityQueueStrategy s p
  => s -> [Stream p] -> Stream a -> Simulation [Stream a]
```

Реализация второй функции приведена ниже для демонстрации подхода. Только нам нужна дополнительная функция, которая бы создавала новый поток как результат повторяющегося исполнения некоторого процесса.

```
repeatProcess :: Process a -> Stream a
```

Вот и сама функция `splitStreamQueueing`:

```
splitStreamQueueing s n x =
  do ref <- liftIO $ newIORef x
     res <- newResource s 1
     let reader =
           usingResource res $
           do p <- liftIO $ readIORef ref
              (a, xs) <- runStream p
              liftIO $ writeIORef ref xs
              return a
     return $ map (\i -> repeatProcess reader) [1..n]
```

Основная идея заключается в том, что многие модели могут быть определены как сеть вычислений `Stream`.

Такая сеть должна иметь внешние входные потоки, обычно случайные потоки подобные следующим.

```
randomUniformStream :: Double -> Double -> Stream (Arrival Double)
randomNormalStream :: Double -> Double -> Stream (Arrival Double)
randomExponentialStream :: Double -> Stream (Arrival Double)
randomErlangStream :: Double -> Int -> Stream (Arrival Double)
randomPoissonStream :: Double -> Stream (Arrival Int)
randomBinomialStream :: Double -> Int -> Stream (Arrival Int)
```

Здесь значение типа `Arrival a` содержит модельное время, при котором возникло внешнее событие, а также содержит само событие типа `a` и задержку по времени, которая прошла с момента прихода предыдущего события.

Для параллельной обработки входного потока мы разбиваем такой поток с помощью функции `splitStream`, обрабатываем параллельно новые потоки, а затем соединяем промежуточные результаты в один выходной поток, используя функцию `concatStreams`. Далее будет приведена функция `processorParallel`, которая делает именно это.

Чтобы обработать заданный поток последовательно некоторыми обслуживающими серверами, нам нужна вспомогательная функция, которая бы читала на одно значение данных больше про запас, играя роль такого промежуточного буфера между серверами.

```
prefetchStream :: Stream a -> Stream a
```

Теперь нам нужна движущая сила, что запустила бы целую сеть потоков.

```
sinkStream :: Stream a -> Process ()
```

Эта функция безостановочно читает данные из входного потока. Это подобно блоку терминатора в GPSS.

6.3 Пассивные потоки и активные сигналы

По крайней мере, существует два разных типа источников данных: потоки и сигналы.

Данные могут быть запрошены явно. Если мы не запрашиваем их, то они и не приходят. Это то, что задает вычисление `Stream`. Только рассмотренные

в предыдущем разделе случайные потоки проверяют, что данные запрашиваются непрерывно и по порядку. Если случайный поток запрошен неверно, то будет ошибка времени исполнения. Это что-то вроде санитарной проверки на то, что модель не содержит явных логических ошибок. В целом же, потоки сами по себе не имеют такой проверки.

Поэтому вам следует быть очень внимательными при использовании вычисления `Stream`. Поток должен непрерывно запрашиваться. Либо вам следует использовать терминатор `sinkStream`, либо перекладывать элементы в очередь, либо исключать элементы из имитации, например, когда очередь заполнена.

Вероятно, GPSS-подобный предметно-ориентированный язык, описанный в разделе 10, является более надежным и простым в использовании, так как он не имеет этих ограничений. Также тот язык очень выразителен. Только он все же требует потоков при определении генераторов.

Тем не менее, потоки могут быть полезны при моделировании многих *сетей очередей*. Например, естественно представлять приход в модель заявок как вычисление `Stream`.

В отличие от потоков, данные в сигналах приходят независимо от того, обрабатываем мы их или нет. Мы можем подписаться на обработку значений `Signal`, и тогда мы будем получать эти значения. Если мы не подписываемся, то сигнал испускает свои значения все равно. Поэтому мы можем сказать, что вычисление `Signal` активно, тогда как вычисление `Stream` пассивно.

Например, мы можем моделировать сигнал WiFi как вычисление `Signal`. Это вычисление может быть также полезным и в других случаях.

6.4 Процессор

Имея некоторый поток данных, было бы естественно оперировать его преобразованием, которое мы назовем *процессором*:

```
newtype Processor a b =
  Processor { runProcessor :: Stream a -> Stream b }
```

Здесь выбор такого названия не был случайным [4]. Этот тип, возможно, является стрелкой `Arrow`, а стрелка может быть интерпретирована как некоторый вид процессора. Только код, вероятно, будет медленным, если мы решим использовать нотацию-*proc*, так как реализация `Arrow` для этого частного типа далеко не оптимальна.

Мы можем создавать процессоры непосредственно из потоков. Опуская очевидные случаи, мы рассмотрим только наиболее важные.

Новый процессор может быть создан по заданной функции обработчика, которая возвращает вычисление `Process`.

```
arrProcessor :: (a -> Process b) -> Processor a b
```

Также мы можем использовать аккумулятор для хранения промежуточного состояния процессора. Во время обработки входного потока и порождения выходного мы можем менять состояние.

```
accumProcessor :: (acc -> a -> Process (acc, b)) -> acc -> Processor a b
```

Произвольное число процессоров может быть объединено для параллельной обработки, используя стратегию очереди `FCFS`:

```
processorParallel :: [Processor a b] -> Processor a b
```

Реализация этой функции основана на использовании рассмотренных ранее мультиплексирующей и демультиплексирующей функциях. Мы разбиваем входной поток, обрабатываем промежуточные потоки параллельно, а потом соединяем результирующие потоки в один выходной поток.

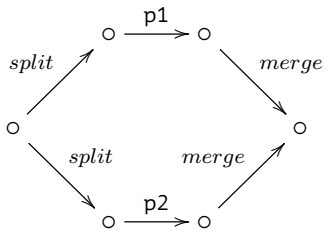
Существуют другие версии функции `processParallel`, где мы можем задать стратегии очереди и приоритеты при необходимости.

Для создания последовательности автономно работающих процессоров мы можем также использовать рассмотренную раннее функцию предварительной обработки:

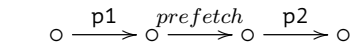
```
prefetchProcessor :: Processor a a
prefetchProcessor = Processor prefetchStream
```

Например, по заданным двум дополняющим друг друга процессорам `p1` и `p2` мы можем создать два новых процессора, где один будет подразумевать параллельную обработку, а другой — последовательную обработку:

```
pPar = processorParallel [p1, p2]
```



```
pSeq = p1 >>> prefetchProcessor >>> p2
```



На основе этого подхода мы можем моделировать на языке Haskell довольно сложные сети очередей простым и высокоуровневым декларативным образом.

Касательно самих очередей, мы можем моделировать их, используя довольно общие вспомогательные процессоры, подобные этому:

```
queueProcessor :: (a -> Process ())
                  -- ^ положить в очередь входной элемент
                  -> Process b
                  -- ^  извлечь из очереди выходной элемент
                  -> Processor a b
                  -- ^ буферизующий процессор
```

Идея заключается в том, что есть огромное количество вариантов, как очереди могут быть объединены в сеть. При добавлении элементов мы можем либо ожидать, пока в заполненной очереди не появится место, либо считать такие элементы как потерянные. Мы можем использовать приоритеты для вычислений `Process`, которые добавляют элементы или извлекают. Более того, разные процессы могут добавлять и извлекать элементы одновременно.

Поэтому автор Айвики решил ввести такие достаточно общие функции для моделирования очередей, где детали того, как моделируются очереди, могут быть кратко описаны с помощью определенных ранее комбинаторов подобных `enqueue`, `enqueueOrLost` и `dequeue`. В дистрибутив Айвики включены примеры.

К сожалению, тип `Processor` не является `ArrowLoop` по той же самой причине, по какой монада `Process` не является `MonadFix` — все из-за продолжений. Тем не менее, мы можем моделировать сети очередей с обратными связями, используя промежуточные очереди для задержания потока. Одна из таких функций приведена ниже.

```
queueProcessorLoopSeq ::
  (a -> Process ())
  -- ^ положить в очередь входной элемент
  -> Process c
  -- ^ извлечь из очереди элемент для дальнейшей обработки
  -> Processor c (Either e b)
  -- ^ обработать и решить, нужно ли значение типа @e@
  -- еще обработать в цикле (условие)
```

```
  -> Processor e a
  -- ^ обработать в цикле и затем вернуть некоторое
  -- значение типа @a@ снова в очередь (тело цикла)
  -> Processor a b
  -- ^ буферизующий процессор
```

Далее в разделе 6.7 приведен пример модели, которая использует потоки и процессоры очередей.

Вероятно, что мой читатель найдет для себя GPSS-подобный предметно-ориентированный язык, рассматриваемый в главе 10, более удобным для использования в своих моделях. Айвика естественным образом поддерживает этот встроенный язык. Он также основан на вычислении `Process` и хорошо интегрируется со всем, что здесь описано.

Используя процессоры, мы можем моделировать довольно сложное поведение. Например, мы можем моделировать стратегию Round-Robbin[17] для обработки.

```
roundRobbinProcessor :: Processor (Process Double, Process a) a
```

Функция пытается выполнить задачу в течение заданного тайм-аута. Если задача занимает больше времени, то она отменяется, и мы снова возвращаемся к процессору. Иначе, в случае успеха результат передается на вывод. Тайм-аут и задача передаются процессору из входного потока.

Процессоры и потоки позволяют создавать процесс-ориентированные модели высокого уровня способом, который близок к тому, что описан в книге A. Alan B. Pritsker и Jean J. O'Reilly [12].

В то же самое время, все вычисления хорошо интегрированы в Айвике, и мы можем комбинировать разные походы в рамках одной модели, например, комбинировать процесс-ориентированную модель и агентную модель вместе.

6.5 Сервер

В Айвике существует тип данных `Server`, который позволяет моделировать рабочее место, и который также собирает соответствующую статистику во время имитации.

```
data Server s a b

newServer :: (a -> Process b) -> Simulation (Server () a b)
```

```
newStateServer :: (s -> a -> Process (s, b))
                  -> s
                  -> Simulation (Server s a b)
```

Для создания сервера мы предоставляем обрабатывающую функцию, которая принимает входное значение, обрабатывает его и порождает вывод внутри вычисления `Process`. Обрабатывающая функция может использовать аккумулятор для сохранения состояния сервера во время обработки.

Чтобы включить сервер в имитацию, мы можем использовать его процессор, который выполняет обслуживание и обновляет внутренние счетчики.

```
serverProcessor :: Server s a b -> Processor a b
```

Например, при подготовке результатов моделирования к выводу мы можем запросить статистику по времени, проведенному сервером во время обработки задач.

```
serverProcessingTime :: Server s a b -> Event (SamplingStats Double)
```

Есть один тонкий момент. Каждый раз, как мы используем результат функции `serverProcessor`, то мы в действительности создаем новый процессор, который ссылается на тот же самый сервер, а следовательно, этот процессор обновляет те же самые счетчики статистики. Это может быть полезно, если мы собираемся собрать статистику для группы серверов, работающих параллельно, хотя лучшей практикой было бы все же использовать функцию `serverProcessor` только один раз для каждого отдельно взятого сервера.

Для сброса статистики по серверу в текущем модельном времени мы можем вызвать следующую функцию:

```
resetServer :: Server s a b -> Event ()
```

Мы можем запросить свойства сервера в имитационном эксперименте подобно тому, как мы запрашивали свойства ресурса в разделе 3.6.

```
import qualified Simulation.Aivika.Results.Transform as T

s = T.Server $ resultByName "someServer"
k = T.tr $ T.serverProcessingTime s
```

6.6 Измерение времени обработки

Чтобы измерить время обработки заявок, в примерах часто используется простой объект, который называется `ArrivalTimer`.

```
data ArrivalTimer

newArrivalTimer :: Simulation ArrivalTimer

arrivalTimerProcessor :: ArrivalTimer
                      -> Processor (Arrival a) (Arrival a)

arrivalProcessingTime :: ArrivalTimer
                      -> Event (SamplingStats Double)
```

Каждый раз, как мы применяем во время имитации результат функции `arrivalTimerProcessor`, соответствующий внутренний счетчик обновляется для того, чтобы измерить время обработки заявок, которые содержат точное время входа в имитационную модель. Затем этот счетчик возвращается в виде сводной статистики функцией `arrivalProcessingTime`.

Обычно же время обработки запрашивается в рамках имитационного эксперимента с помощью кода похожего на этот:

```
import qualified Simulation.Aivika.Results.Transform as T

t = T.ArrivalTimer $ resultByName "someTimer"
k = T.tr $ T.arrivalProcessingTime t
```

6.7 Пример: сеть очередей

Для иллюстрации того, как потоки и процессоры могут быть использованы при моделировании, давайте рассмотрим модель [12, 17] производственной линии с пунктами технического контроля и настройки. Эта модель имеет поток обработки с циклом. Есть две бесконечные очереди.

> Собранные телевизоры на заключительной стадии производства проходят ряд пунктов технического контроля. В последнем из этих пунктов проверяется настройка телевизоров. Если при проверке обнаружилось, что телевизор работает некачественно, он направляется в пункт настройки, где настраивается заново. После перенастройки телевизор снова направляется в последний пункт контроля для проверки качества настройки. Телевизоры, которые сразу

или после нескольких возвратов в пункт настройки прошли фазу заключительной проверки, направляются в цех упаковки.

Время между поступлениями телевизоров в пункт контроля для заключительной проверки распределено равномерно на интервале 3,5 - 7,5 минут. В пункте заключительной проверки параллельно работают два контролера. Время, необходимое на проверку одного телевизора, распределено равномерно на интервале 6 - 12 минут. В среднем, 85% телевизоров проходят проверку успешно и направляются на упаковку. Остальные 15% возвращаются в пункт настройки, обслуживаемый одним рабочим. Время настройки распределено равномерно на интервале 20 - 40 минут.

Необходимо сымитировать работу пунктов контроля и настройки в течение 480 минут для оценки времени, затрачиваемого на обслуживание каждого телевизора на последнем этапе производства, а также загрузки контролеров и настройщика.

6.7.1 Возвращение результатов из модели

Ниже приведена модель, которая определена достаточно декларативным и прямолинейным способом.

```
module Model (model) where

import Prelude hiding (id, (.))

import Control.Monad
import Control.Monad.Trans
import Control.Arrow
import Control.Category (id, (.))

import Simulation.Aivika
import Simulation.Aivika.Queue.Infinite

-- минимальная задержка прихода следующего телевизора
minArrivalDelay = 3.5

-- максимальная задержка прихода следующего телевизора
maxArrivalDelay = 7.5

-- минимальное время проверки телевизора
minInspectionTime = 6
```

```
-- максимальное время проверки телевизора
maxInspectionTime = 12

-- вероятность прохождения проверки
inspectionPassingProb = 0.85

-- как много проверяющих контролеров?
inspectionStationCount = 2

-- минимальное время перенастройки телевизора
minAdjustmentTime = 20

-- максимальное время перенастройки телевизора
maxAdjustmentTime = 40

-- как много работников, занимающихся перенастройкой?
adjustmentStationCount = 1

-- создать пункт контроля (сервер)
newInspectionStation =
  newServer $ \a ->
  do holdProcess =<<
       (liftParameter $
        randomUniform minInspectionTime maxInspectionTime)
     passed <-
       liftParameter $
       randomTrue inspectionPassingProb
     if passed
       then return $ Right a
       else return $ Left a

-- создать пункт перенастройки (сервер)
newAdjustmentStation =
  newServer $ \a ->
  do holdProcess =<<
       (liftParameter $
        randomUniform minAdjustmentTime maxAdjustmentTime)
     return a

model :: Simulation Results
model = mdo
  -- для подсчета прибывших телевизоров на проверку и настройку
  inputArrivalTimer <- newArrivalTimer
  -- для статистики по времени обработки
  outputArrivalTimer <- newArrivalTimer
```

```
-- определить поток входных событий
let inputStream =
      randomUniformStream minArrivalDelay maxArrivalDelay
-- создать очередь перед пунктами контроля
inspectionQueue <-
  runEventInStartTime newFCFSQueue
-- создать очередь перед пунктами перенастройки
adjustmentQueue <-
  runEventInStartTime newFCFSQueue
-- создать пункты контроля (серверы)
inspectionStations <-
  forM [1 .. inspectionStationCount] $ \_ ->
  newInspectionStation
-- создать пункты перенастройки (серверы)
adjustmentStations <-
  forM [1 .. adjustmentStationCount] $ \_ ->
  newAdjustmentStation
-- цикл процессора для очереди контроля
let inspectionQueueProcessorLoop =
      queueProcessorLoopSeq
      (liftEvent . enqueue inspectionQueue)
      (dequeue inspectionQueue)
      inspectionProcessor
      (adjustmentQueueProcessor >>> adjustmentProcessor)
-- процессор для очереди перенастройки
let adjustmentQueueProcessor =
      queueProcessor
      (liftEvent . enqueue adjustmentQueue)
      (dequeue adjustmentQueue)
-- параллельная работа пунктов контроля
let inspectionProcessor =
      processorParallel (map serverProcessor inspectionStations)
-- возможная параллельная работа пунктов перенастройки
let adjustmentProcessor =
      processorParallel (map serverProcessor adjustmentStations)
-- итоговый процессор от входных данных к выходным
let entireProcessor =
      arrivalTimerProcessor inputArrivalTimer >>>
      inspectionQueueProcessorLoop >>>
      arrivalTimerProcessor outputArrivalTimer
-- запустить имитационную модель
runProcessInStartTime $
  sinkStream $ runProcessor entireProcessor inputStream
-- вернуть результаты моделирования в начальное время
return $
  results
```

```
    [resultSource
     "inspectionQueue" "the inspection queue"
     inspectionQueue,
     --
     resultSource
     "adjustmentQueue" "the adjustment queue"
     adjustmentQueue,
     --
     resultSource
     "inputArrivalTimer" "the input arrival timer"
     inputArrivalTimer,
     --
     resultSource
     "outputArrivalTimer" "the output arrival timer"
     outputArrivalTimer,
     --
     resultSource
     "inspectionStations" "the inspection stations"
     inspectionStations,
     --
     resultSource
     "adjustmentStations" "the adjustment stations"
     adjustmentStations]
```

6.7.2 Задание эксперимента

Модель возвращается несколько источников данных. Мы создаем эксперимент, который показывает данные с разных ракурсов. Названия должны говорить сами за себя.

```
module Experiment (experiment, generators) where

import Data.Monoid

import Control.Arrow

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

import qualified Simulation.Aivika.Results.Transform as T

-- | The simulation specs.
specs = Specs { spcStartTime = 0.0,
                spcStopTime = 480.0,
```

```
                spcDT = 0.1,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

-- | The experiment.
experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1000,
    -- experimentRunCount = 10,
    experimentTitle = "Inspection and Adjustment Stations on " ++
                      "a Production Line (the Monte-Carlo simulation)" }

inputArrivalTimer  = T.ArrivalTimer $ resultByName "inputArrivalTimer"
outputArrivalTimer = T.ArrivalTimer $ resultByName "outputArrivalTimer"

inspectionStations = T.Server $ resultByName "inspectionStations"
adjustmentStations = T.Server $ resultByName "adjustmentStations"

inspectionQueue = T.Queue $ resultByName "inspectionQueue"
adjustmentQueue = T.Queue $ resultByName "adjustmentQueue"

resultProcessingTime :: ResultTransform
resultProcessingTime =
  (T.tr $ T.arrivalProcessingTime inputArrivalTimer) <>
  (T.tr $ T.arrivalProcessingTime outputArrivalTimer)

resultProcessingFactor :: ResultTransform
resultProcessingFactor =
  (T.serverProcessingFactor inspectionStations) <>
  (T.serverProcessingFactor adjustmentStations)

inspectionQueueCount      = T.queueCount inspectionQueue
inspectionQueueCountStats = T.tr $ T.queueCountStats inspectionQueue
inspectionWaitTime        = T.tr $ T.queueWaitTime inspectionQueue

adjustmentQueueCount      = T.queueCount adjustmentQueue
adjustmentQueueCountStats = T.tr $ T.queueCountStats adjustmentQueue
adjustmentWaitTime        = T.tr $ T.queueWaitTime adjustmentQueue

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView defaultInfoView,
   outputView $ defaultFinalStatsView {
```

```
    finalStatsTitle  = "Arrivals",
    finalStatsSeries = resultProcessingTime },
  outputView $ defaultDeviationChartView {
    deviationChartTitle = "The processing factor (chart)",
    deviationChartWidth = 1000,
    deviationChartRightYSeries = resultProcessingFactor },
  outputView $ defaultFinalHistogramView {
    finalHistogramTitle = "The processing factor (histogram)",
    finalHistogramWidth = 1000,
    finalHistogramSeries = resultProcessingFactor },
  outputView $ defaultFinalStatsView {
    finalStatsTitle = "The processing factor (statistics)",
    finalStatsSeries = resultProcessingFactor },
  outputView $ defaultDeviationChartView {
    deviationChartTitle = "The inspection queue size (chart)",
    deviationChartWidth = 1000,
    deviationChartRightYSeries =
      inspectionQueueCount <> inspectionQueueCountStats },
  outputView $ defaultFinalStatsView {
    finalStatsTitle = "The inspection queue size (statistics)",
    finalStatsSeries = inspectionQueueCountStats },
  outputView $ defaultDeviationChartView {
    deviationChartTitle = "The inspection queue wait time (chart)",
    deviationChartWidth = 1000,
    deviationChartRightYSeries = inspectionWaitTime },
  outputView $ defaultFinalStatsView {
    finalStatsTitle = "The inspection queue wait time (statistics)",
    finalStatsSeries = inspectionWaitTime },
  outputView $ defaultDeviationChartView {
    deviationChartTitle = "The adjustment queue size (chart)",
    deviationChartWidth = 1000,
    deviationChartRightYSeries =
      adjustmentQueueCount <> adjustmentQueueCountStats },
  outputView $ defaultFinalStatsView {
    finalStatsTitle = "The adjustment queue size (statistics)",
    finalStatsSeries = adjustmentQueueCountStats },
  outputView $ defaultDeviationChartView {
    deviationChartTitle = "The adjustment queue wait time (chart)",
    deviationChartWidth = 1000,
    deviationChartRightYSeries = adjustmentWaitTime },
  outputView $ defaultFinalStatsView {
    finalStatsTitle = "The adjustment queue wait time (statistics)",
    finalStatsSeries = adjustmentWaitTime } ]
```

6.7.3 Вывод графиков

Здесь мы можем выбрать один из интерфейсов графиков. Код абсолютно такой же, каким он был в разделе 1.4.3.

6.7.4 Запуск имитационного эксперимента

На моем ноутбуке заданный имитационный эксперимент длился 15 секунд при использовании графического интерфейса на основе библиотеки Cairo, а при использовании интерфейса на основе библиотеки Diagrams он длился 28 секунд.

Вы можете увидеть один из полученных графиков на рисунке 6.1.

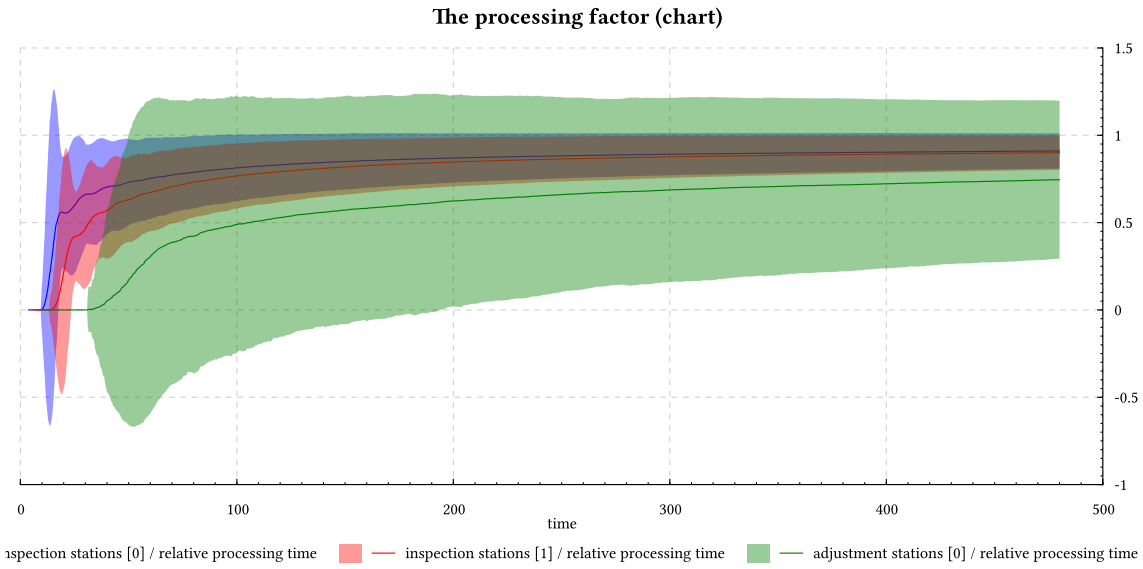


Рис. 6.1: График отклонения для загрузки контролеров и настройщика.

6.8 Пример: вытеснение ресурса

Наступило время выполнить обещание. Ниже представлен пример, который демонстрирует использование вытеснения ресурса. Этот пример моделирует работу станка с поломками [12, 17].

Задания поступают на станок в среднем 1 раз в час. Распределение величины интервала между ними экспоненциально. При нормальном режиме работы задания выполняются в порядке их поступления. Время выполнения задания нормально распределено с математическим ожиданием 0,5 часа и среднеквадратичным отклонением 0,1. Перед выполнением задания производится наладка станка, время которой распределено равномерно на интервале 0,2 - 0,5 часа. Задания, выполненные на станке, направляются в другие отделы цеха и считаются покинувшими рассматриваемую систему.

Станок время от времени ломается. Интервалы между поломками распределены нормально с математическим ожиданием 20 часов и среднеквадратичным отклонением 2 часа. При поломке выполняемое задание удаляется со станка и помещается в начало очереди заданий к станку. Выполнение задания возобновляется с того места, на котором оно было прервано.

Когда станок ломается, начинается процесс устранения неисправности, который состоит из трех фаз. Продолжительность каждой фазы распределена экспоненциально с математическим ожиданием, равным $3/4$ часа. Поскольку общая продолжительность устранения поломки является суммой независимых и одинаково распределенных случайных величин с одинаковыми параметрами, она имеет эрланговское распределение.

Работа станка анализируется в течение 500 часов для получения информации о загрузке станка и времени выполнения задания. Статистика собирается в течение тысячи имитационных запусков.

6.8.1 Возвращение результатов из модели

Как и прежде, мы определяем модель, которая возвращает результаты моделирования. Здесь мы используем бесконечную очередь, а также вытеснение ресурса, рассмотренное ранее в разделе 3.8.

```
module Model (model) where

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

```
import Data.Monoid
import Data.List

import Simulation.Aivika
import qualified Simulation.Aivika.Queue.Infinite as IQ
import qualified Simulation.Aivika.Resource.Preemption as PR

-- | Как часто приходят задания на станок (экспоненциально)?
jobArrivingMu = 1

-- | Среднее время обработки задания (нормально).
jobProcessingMu = 0.5

-- | Среднеквадратичное отклонение времени обработки задания (нормально).
jobProcessingSigma = 0.1

-- | Минимальное время наладки станка (равномерное).
minSetUpTime = 0.2

-- | Максимальное время наладки станка (равномерное).
maxSetUpTime = 0.5

-- | Среднее время между поломками (нормальное).
breakdownMu = 20

-- | Среднеквадратичное отклонение времени между поломками (нормальное).
breakdownSigma = 2

-- | Среднее время каждой из трех фаз восстановления (эрланговское).
repairMu = 3/4

-- | Приоритет заданий (меньше - выше)
jobPriority = 1

-- | Приоритет поломки (меньше - выше)
breakdownPriority = 0

-- | Имитационная модель.
model :: Simulation Results
model = do
  -- создать входную очередь
  inputQueue <- runEventInStartTime IQ.newFCFSQueue
  -- счетчик выполненных заданий
  jobsCompleted <- newArrivalTimer
  -- счетчик прерванных заданий
  jobsInterrupted <- newRef (0 :: Int)
```

```
-- создать входной поток
let inputStream =
      randomExponentialStream jobArrivingMu
-- создать вытесняемый ресурс
tool <- runEventInStartTime $ PR.newResource 1
-- наладка станка
machineSettingUp <-
  newPreemptibleRandomUniformServer
  True minSetUpTime maxSetUpTime
-- обработка заданий на станке
machineProcessing <-
  newPreemptibleRandomNormalServer
  True jobProcessingMu jobProcessingSigma
-- поломка станка
let machineBreakdown =
      do randomNormalProcess_ breakdownMu breakdownSigma
         PR.usingResourceWithPriority tool breakdownPriority $
           randomErlangProcess_ repairMu 3
         machineBreakdown
-- запустить процесс поломок
runProcessInStartTime machineBreakdown
-- обновлять счетчики при прерывании заданий
runEventInStartTime $
  handleSignal_
  (serverTaskPreemptionBeginning machineProcessing) $ \a ->
  modifyRef jobsInterrupted (+ 1)
-- определить сеть очередей
let network =
      queueProcessor
      (\a -> liftEvent $ IQ.enqueue inputQueue a)
      (IQ.dequeue inputQueue) >>>
      (withinProcessor $
       PR.requestResourceWithPriority tool jobPriority) >>>
      serverProcessor machineSettingUp >>>
      serverProcessor machineProcessing >>>
      (withinProcessor $ PR.releaseResource tool) >>>
      arrivalTimerProcessor jobsCompleted
-- запустить станок
runProcessInStartTime $
  sinkStream $ runProcessor network inputStream
-- вернуть результаты моделирования в начальное время
return $
  results
  [resultSource
   "inputQueue" "the queue of jobs"
   inputQueue,
```

```
    --
    resultSource
    "machineSettingUp" "the machine setting up"
    machineSettingUp,
    --
    resultSource
    "machineProcessing" "the machine processing"
    machineProcessing,
    --
    resultSource
    "jobsInterrupted" "a counter of the interrupted jobs"
    jobsInterrupted,
    --
    resultSource
    "jobsCompleted" "a counter of the completed jobs"
    jobsCompleted,
    --
    resultSource
    "tool" "the machine tool"
    tool]
```

6.8.2 Задание эксперимента

Имитационный эксперимент определяет 1000 запусков. Нам интересны время обработки, время ожидания в очереди, размер очереди, и коэффициент загрузки станка, который здесь обозначается как коэффициент времени обработки сервером или относительное время обработки.

```
module Experiment (experiment, generators) where

import Data.Monoid

import Control.Arrow

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

import qualified Simulation.Aivika.Results.Transform as T

-- | Параметры моделирования.
specs = Specs { spcStartTime = 0.0,
                spcStopTime = 500.0,
                spcDT = 0.1,
```

```
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

-- | Эксперимент.
experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1000,
    -- experimentRunCount = 10,
    experimentTitle = "Machine Tool with Breakdowns" }

jobsCompleted     = T.ArrivalTimer $ resultByName "jobsCompleted"
jobsInterrupted   = resultByName "jobsInterrupted"
inputQueue        = T.Queue $ resultByName "inputQueue"
machineProcessing = T.Server $ resultByName "machineProcessing"

jobsCompletedCount =
  T.samplingStatsCount $
  T.arrivalProcessingTime jobsCompleted

processingTime :: ResultTransform
processingTime =
  T.tr $ T.arrivalProcessingTime jobsCompleted

waitTime :: ResultTransform
waitTime =
  T.tr $ T.queueWaitTime inputQueue

queueCount :: ResultTransform
queueCount =
  T.queueCount inputQueue

queueCountStats :: ResultTransform
queueCountStats =
  T.tr $ T.queueCountStats inputQueue

processingFactor :: ResultTransform
processingFactor =
  T.serverProcessingFactor machineProcessing

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView defaultInfoView,
   outputView $ defaultFinalStatsView {
```

```
    finalStatsTitle  = "Machine Tool With Breakdowns",
    finalStatsSeries = jobsCompletedCount <> jobsInterrupted },
  outputView $ defaultDeviationChartView {
    deviationChartTitle = "The Wait Time (chart)",
    deviationChartWidth = 1000,
    deviationChartRightYSeries = waitTime },
  outputView $ defaultFinalStatsView {
    finalStatsTitle = "The Wait Time (statistics)",
    finalStatsSeries = waitTime },
  outputView $ defaultDeviationChartView {
    deviationChartTitle = "The Queue Size (chart)",
    deviationChartWidth = 1000,
    deviationChartRightYSeries = queueCount <> queueCountStats },
  outputView $ defaultFinalStatsView {
    finalStatsTitle = "The Queue Size (statistics)",
    finalStatsSeries = queueCountStats },
  outputView $ defaultDeviationChartView {
    deviationChartTitle = "The Processing Time (chart)",
    deviationChartWidth = 1000,
    deviationChartRightYSeries = processingTime },
  outputView $ defaultFinalStatsView {
    finalStatsTitle = "The Processing Time (statistics)",
    finalStatsSeries = processingTime },
  outputView $ defaultDeviationChartView {
    deviationChartTitle = "The Machine Load (chart)",
    deviationChartWidth = 1000,
    deviationChartRightYSeries = processingFactor },
  outputView $ defaultFinalHistogramView {
    finalHistogramTitle = "The Machine Load (histogram)",
    finalHistogramWidth = 1000,
    finalHistogramSeries = processingFactor },
  outputView $ defaultFinalStatsView {
    finalStatsTitle = "The Machine Load (statistics)",
    finalStatsSeries = processingFactor } ]
```

6.8.3 Вывод графиков

Как это стало традицией, мы здесь можем выбрать один из интерфейсов графиков. Код абсолютно такой же, каким он был в разделе 1.4.3.

6.8.4 Запуск имитационного эксперимента

При использовании интерфейса графиков на основе Cairo имитационный эксперимент с 1000-ю запусками длился 12 секунд на ноутбуке автора. Однако, при использовании интерфейса на основе библиотеки Diagrams, тот же самый эксперимент длился 22 секунды.

На гистограмме из рисунка 6.2, мы можем видеть приближение для распределения коэффициента загрузки станка в конечной точке времени.

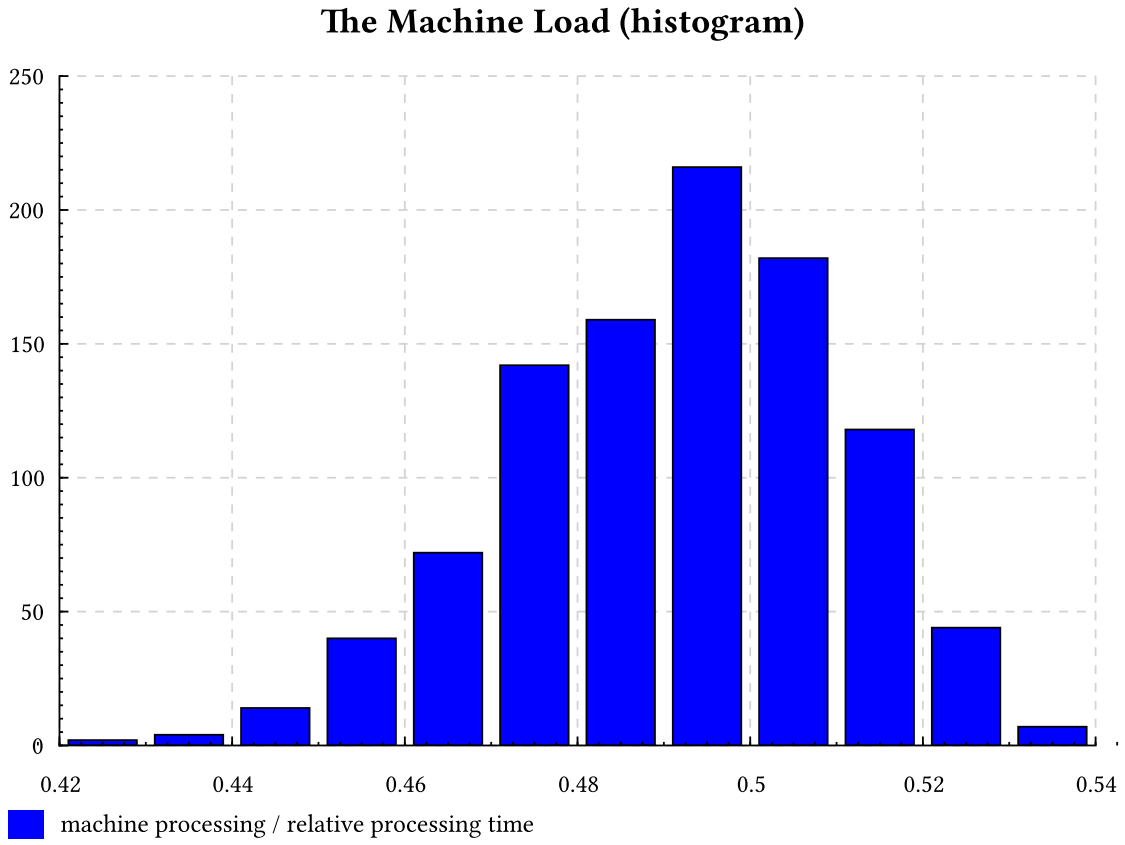


Рис. 6.2: Гистограмма коэффициента загрузки станка в конечной точке времени.

Глава 7

Агентное моделирование

Айвика предоставляет базовую поддержку парадигмы агентного моделирования [16].

Идея заключается в том, что мы пытаемся описать модель как совместное поведение относительно большого числа небольших агентов. Агенты могут иметь состояния, и эти состояния могут быть либо активными, либо неактивными. Мы можем назначить состоянию обработчик, который исполнится при условии, что состояние останется активным.

7.1 Агенты и их состояния

Мы создаем новые агенты и их состояния в рамках вычисления `Simulation`.

```
data Agent
data AgentState

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

Только одно из состояний может быть выбрано для каждого агента в точке модельного времени. Все предки состояния остаются активными, если они были активными до этого, или они становятся активными, если были неактивными. Напротив, другие состояния становятся неактивными, если они были активными.

```
selectedState :: Agent -> Event (Maybe AgentState)
selectState :: AgentState -> Event ()
```

Первая функция возвращает текущее выбранное состояние или `Nothing`, если агент еще не был инициализирован. Другая функция позволяет выбрать новое состояние. Обе функции возвращают действия внутри вычисления `Event`, что означает, что выбор состояние всегда синхронизирован с очередью событий.

Мы можем назначить обработчики `Event`, которые будут исполнены во время активации и деактивации заданного состояния во время подобного выбора.

```
setStateActivation :: AgentState -> Event () -> Event ()
setStateDeactivation :: AgentState -> Event () -> Event ()
```

Если заданное третье состояние остается активным при выборе другого состояния, а путь от старого выбранного состояния к новому проходит через третье состояние, то тогда мы можем назвать третье состояние *переходным*, и можем назначить ему действие, которое будет выполнено, если подобный переход произойдет.

```
setStateTransition :: AgentState -> Event (Maybe AgentState) -> Event ()
```

Здесь новое выбранное состояние посылается соответствующему вычислению `Event`.

Что отличает агенты от других концепций моделирования, так это возможность назначать обработчики *тайм-аута* и *таймера*. Обработчик тайм-аута — это такое вычисление `Event`, которое исполнится через заданный промежуток времени, если состояние останется активным. Обработчик таймера похож, но только обработчик повторяется, пока состояние все еще остается активным. Поэтому обработчик тайм-аута принимает время как чистое значение, тогда как обработчик таймера заново вычисляет промежуток времени внутри вычисления `Event` после каждого успешного исполнения.

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

Реализация довольно проста. По заданному обработчику состояния, мы создаем обработчик-оболочку, которую передаем функции `enqueueEvent` с желаемым временем активации. Если состояние становится неактивным до истечения запланированного времени, то тогда мы аннулируем обработчик-оболочку. После того, как эта оболочка вызовется через очередь событий в запланированное время, мы просто не вызываем соответствующий обработчик состояния, если обработчик-оболочка была ранее аннулирована.

Мы используем вычисление `Event` для синхронизации агентов с очередью событий. Это буквально означает, что агентное моделирование может быть интегрировано с другими имитационными методами в рамках одной комбинированной модели.

7.2 Пример: агентная модель

Чтобы показать, как можно использовать агенты, давайте возьмем модель распространения продукта по Бассу из документации к AnyLogic [16].

> Модель Басса описывает процесс распространения продукта. Изначально продукт никому не известен, и для того, чтобы люди начали его приобретать, он рекламируется. В итоге определенная доля людей приобретает продукт под воздействием рекламы. Также люди приобретают продукт в результате общения с теми, кто этот продукт уже приобрел. Процесс приобретения нового продукта под влиянием убеждения его владельцев чем-то похож на распространение эпидемии.

7.2.1 Возвращение результатов из модели

Ниже приведена имитационная модель. Агенты довольно просты. Они могут находиться только в одном из двух возможных состояний.

```
module Model (model) where

import Data.Array

import Control.Monad
import Control.Monad.Trans

import Simulation.Aivika

n = 100    -- число агентов

advertisingEffectiveness = 0.011
contactRate = 100.0
adoptionFraction = 0.015

data Person = Person { personAgent :: Agent,
```

```
                      personPotentialAdopter :: AgentState,
                      personAdopter :: AgentState }

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

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

definePerson :: Person
                -> Array Int Person
                -> Ref Int
                -> Ref Int
                -> Event ()
definePerson p ps potentialAdopters adopters =
  do setStateActivation (personPotentialAdopter p) $
       do modifyRef potentialAdopters $ \a -> a + 1
          -- добавить тайм-аут
          t <- liftParameter $
               randomExponential (1 / advertisingEffectiveness)
          let st  = personPotentialAdopter p
              st' = personAdopter p
          addTimeout st t $ selectState st'
     setStateActivation (personAdopter p) $
       do modifyRef adopters  $ \a -> a + 1
          -- добавить таймер, который будет работать,
          -- пока активно состояние
          let t = liftParameter $
                  randomExponential (1 / contactRate)    -- много раз!
          addTimer (personAdopter p) t $
            do i <- liftParameter $
                    randomUniformInt 1 n
               let p' = ps ! i
               st <- selectedState (personAgent p')
               when (st == Just (personPotentialAdopter p')) $
                 do b <- liftParameter $
```

```
                        randomTrue adoptionFraction
                    when b $ selectState (personAdopter p')
      setStateDeactivation (personPotentialAdopter p) $
        modifyRef potentialAdopters $ \a -> a - 1
      setStateDeactivation (personAdopter p) $
        modifyRef adopters $ \a -> a - 1

definePersons :: Array Int Person -> Ref Int -> Ref Int -> Event ()
definePersons ps potentialAdopters adopters =
  forM_ (elems ps) $ \p ->
  definePerson p ps potentialAdopters adopters

activatePerson :: Person -> Event ()
activatePerson p = selectState (personPotentialAdopter p)

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

model :: Simulation Results
model =
  do potentialAdopters <- newRef 0
     adopters <- newRef 0
     ps <- createPersons
     runEventInStartTime $
       do definePersons ps potentialAdopters adopters
          activatePersons ps
     return $
       results
       [resultSource
        "potentialAdopters" "potential adopters of the product"
        potentialAdopters,
        resultSource
        "adopters" "adopters of the product" adopters]
```

7.2.2 Задание эксперимента

В нашем имитационном эксперименте мы желаем увидеть график отклонения для числа владельцев и потенциальных покупателей.

```
module Experiment (experiment, generators) where

import Data.Monoid

import Simulation.Aivika
```

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

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

experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1000,
    experimentDescription =
      "This is the famous Bass Diffusion " ++
      "model solved with help of the agent-based modelling." }

potentialAdopters = resultByName "potentialAdopters"
adopters = resultByName "adopters"

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView defaultInfoView,
   outputView $ defaultDeviationChartView {
     deviationChartLeftYSeries =
        potentialAdopters <> adopters } ]
```

7.2.3 Вывод графиков

Как и прежде, здесь мы выбираем один из интерфейсов графиков. Код абсолютно такой же, каким он был в разделе 1.4.3.

7.2.4 Запуск имитационного эксперимента

При использовании интерфейса графиков на основе Diagrams полный имитационный эксперимент с 1000-ю запусками длился 20 секунд на моем ноутбуке. При использовании интерфейса на основе Cairo такой же эксперимент занял только 12 секунд.

Вы можете видеть соответствующий график на рисунке 7.1.

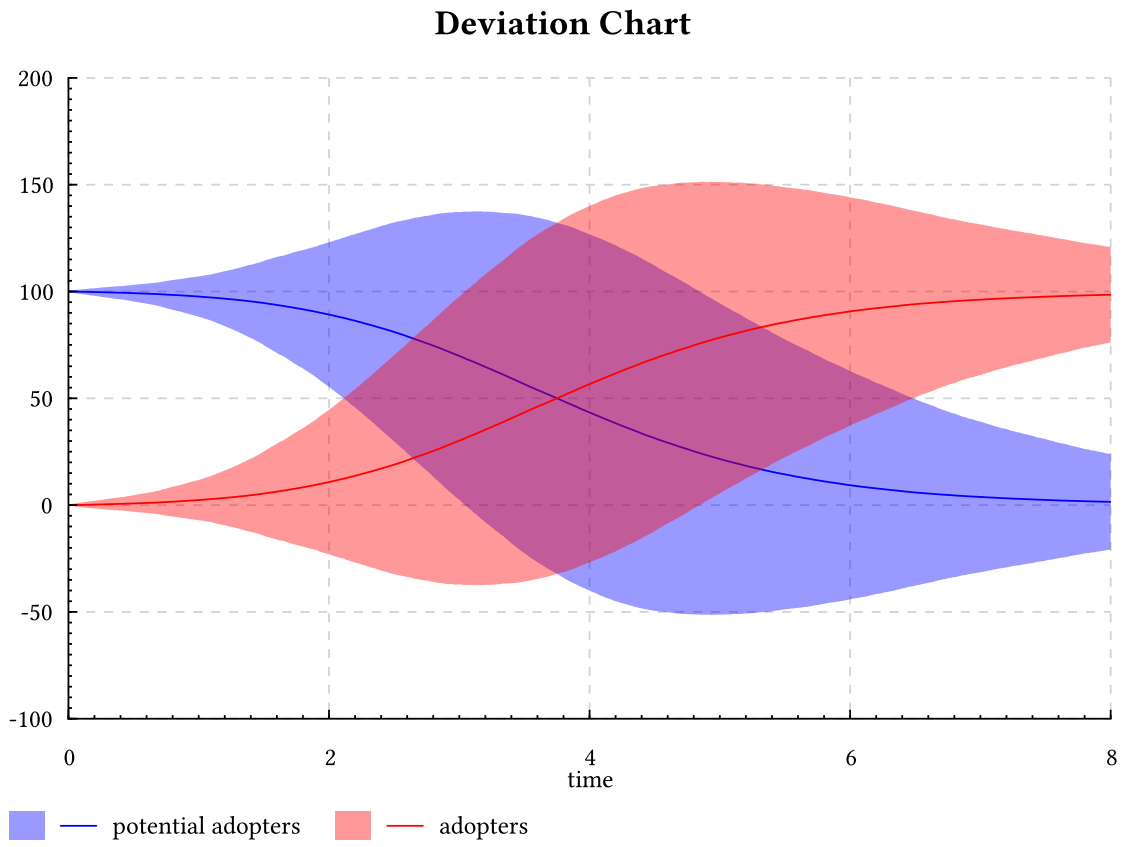


Рис. 7.1: График отклонения для числа владельцев и потенциальных покупателей.

Глава 8

Автоматы

В Айвике существуют два дополнительных вычисления, которые были навеяны идеей автоматов, описанных в литературе [10, 5], что созвучно подходу, применяемому в Yampa [9].

8.1 Схема

Ниже дано первое вычисление, которое называется `Circuit`. Переведем его как *схему*.

```
newtype Circuit a b =
  Circuit { runCircuit :: a -> Event (b, Circuit a b) }
```

Это автомат, который принимает на входе некоторое значение, а затем возвращает вычисленное значение на выходе и следующее состояние внутри вычисления `Event`, синхронизированного с очередью событий.

Очевидно, что тип `Circuit` является `ArrowLoop`. Поэтому мы можем создавать рекурсивные связи и вводить задержки на один такт, используя заданное начальное значение.

```
delayCircuit :: a -> Circuit a a
```

Также мы можем численно интегрировать дифференциальные уравнения в рамках вычисления схемы, создавая при необходимости обратные связи с помощью нотации-*proc*.

```
integCircuit :: Double  -> Circuit Double Double
```

По заданному начальному значению мы возвращаем схему, которая трактует входное значение как производную и возвращает значение интеграла на выходе.

Похожим образом мы можем численно решать системы разностных уравнений, где следующая функция принимает также начальное значение, но возвращает автомат, который генерирует сумму на выходе по заданной на входе разности.

```
sumCircuit :: Num a => a -> Circuit a a
```

Так, система ОДУ из раздела 1.3 может быть переписана следующим образом.

```
circuit :: Circuit () [Double]
circuit =
  let ka = 1
      kb = 1
  in proc () -> do
    rec a  <- integCircuit 100 -< - ka * a
        b  <- integCircuit 0 -< ka * a - kb * b
        c  <- integCircuit 0 -< kb * b
    returnA -< [a, b, c]
```

Для получения конечного результата интегрирования теперь мы должны как-то привести вычисление в стрелке `Circuit` во что-то другое. Поэтому нам нужно некоторое преобразование.

Можно трактовать произвольную схему как преобразование сигналов или как процессор.

```
circuitSignaling :: Circuit a b -> Signal a -> Signal b
circuitProcessor :: Circuit a b -> Processor a b
```

Более того, схема может быть аппроксимирована в точках интегрирования и интерполирована в других точках времени:

```
circuitTransform :: Circuit a b -> Transform a b
```

Здесь тип `Transform` похож на окончание функции `integ`. Поэтому вычисление `Transform` можно воспринимать как аналоговую схему, противопоставляя ее цифровой, которая уже представляется вычислением `Circuit`.

```
newtype Transform a b =
  Transform { runTransform :: Dynamics a -> Simulation (Dynamics b) }
```

Это также дает другую интерпретацию вычисления `Dynamics`. Последнее можно рассматривать как единую сущность, определенную во всех точках времени одновременно, что кажется естественным, так как мы аппроксимируем интеграл таким способом. Сравните это вычисление с вычислением `Event`, где мы строго подчеркиваем то, что вычисление `Event` связано с текущей точкой модельного времени. Говоря же о вычислении `Dynamics`, мы не делаем никаких предположений о модельном времени, кроме краевого условия.

Возвращаясь к нашим ОДУ, мы можем запустить модель, для простоты аппроксимируя схему в точках интегрирования, хотя это будет не очень эффективно.

```
model :: Simulation [Double]
model =
  do results <-
       runTransform (circuitTransform circuit) $
       return ()
     runDynamicsInStopTime results
```

Эта модель вернет почти те же самые результаты[1], но это будет много медленнее той модели, которая использовала функцию `integ` с вычислением `Dynamics`.

Функция `integCircuit` сама по себе имеет очень малое потребление памяти, или говоря точнее, она создает много маленьких краткоживущих объектов, которые почти сразу помечаются как мусор, а следовательно, могут быть удалены. Однако, такое малое потребление уходит на второй план после использования функции `circuitTransform`, которая уже потребляет много памяти.

Напротив, функция `integ` может выделить, в общем-то, довольно большой массив за раз, но потом она почти не потребляет памяти во время численного интегрирования.

Сравнивая схему с другими вычислениями, первая всегда возвращает свои значения на выходе в текущей точке модельного времени без задержки, но схема также сохраняет свое состояние, и мы можем запросить позже следующий вывод по следующему вводу в любое желаемого время. Существенно то, что схема позволяет задавать рекурсивные связи, которые, например, могут быть полезны для описания простых цифровых схем.

[1] Даже если мы будем использовать эквивалентный этому метод Эйлера, все же есть неизбежная погрешность вычислений.

Аппроксимация схемы в точках интегрирования позволяет использовать такую схему в дифференциальных уравнениях. Синхронизация с очередью событий обеспечивается автоматически.

8.2 Сеть

Существует другая версия вычисления `Circuit`, где только вычисление `Event` заменено вычислением `Process`.

```
newtype Net a b = Net { runNet :: a -> Process (b, Net a b) }
```

Тип `Net` имеет более эффективную реализацию класса типов `Arrow`, чем имеет `Processor`. Полученное с помощью нотации-*proc* вычисление должно быть значительно более легковесным. Оно похоже на вычисление `Circuit`, но только `Net` не является `ArrowLoop`, потому что, будучи основанной на продолжениях, монада `Process` не является `MonadFix`.

Более того, вычисление `Net` может быть легко преобразовано в процессор, и это может быть сделано очень эффективно, что показывает главный вариант использования этого типа: написание некоторых частей модели в рамках вычисления `Net`, используя нотацию-*proc*, с последующим преобразованием.

```
netProcessor :: Net a b -> Processor a b
netProcessor = Processor . loop
  where loop x as =
          Cons $
          do (a, as') <- runStream as
             (b, x') <- runNet x a
             return (b, loop x' as')
```

Проблема в том, что тип `Net` не имеет четких возможностей по мультиплексированию и демультиплексированию для имитации параллельной обработки. Также его обратное преобразование довольно дорогое, и в действительности нет гарантии, что заданный процессор будет возвращать в точности одно значение на выходе по каждому входному значению.

```
processorNet :: Processor a b -> Net a b
```

Тем не менее, оба вычисления могут быть полезны в сочетании.

Глава 9

Системная динамика

Ранее мы ввели функцию `integ`, которая позволяет нам аппроксимировать интегралы. Существует похожая функция `diffsum`, которая позволяет определять разностные уравнения. Обыкновенные дифференциальные и разностные уравнения являются основанием системной динамики. В этой главе мы рассмотрим некоторые примеры, относящиеся к этой области моделирования.

9.1 Пример: параметрическая модель

Сейчас мы зададимся следующим вопросом полезным для практики: как подготовить параметрическую модель для имитационного эксперимента по методу Монте-Карло? Например, это может быть полезно для проведения анализа чувствительности.

Для демонстрации давайте возьмем финансовую модель [18], описанную в Vensim 5 Modeling Guide, главе Financial Modeling and Risk. Вероятно, лучшим способом описать модель будет просто привести ее уравнения.

Уравнения используют функцию `npv` из системной динамики. Она возвращает чистую приведенную стоимость (NPV) потока, вычисленного на основе заданных учетной ставки, начального значения и некоторого коэффициента (обычно 1).

```
npv :: Dynamics Double                  -- ^ поток
       -> Dynamics Double               -- ^ учетная ставка
       -> Dynamics Double               -- ^ начальное значение
       -> Dynamics Double               -- ^ коэффициент
       -> Simulation (Dynamics Double)  -- ^ чистая приведенная стоимость
```

```
npv stream rate init factor =
  mdo let dt' = liftParameter dt
      df <- integ (- df * rate) 1
      accum <- integ (stream * df) init
      return $ (accum + dt' * stream * df) * factor
```

Также нам нужен вспомогательный условный комбинатор, который позволит избежать в некоторых случаях использования нотации-*do*.

```
-- | Реализует оператор if-then-else.
ifDynamics :: Dynamics Bool -> Dynamics a -> Dynamics a -> Dynamics a
ifDynamics cond x y =
  do a <- cond
     if a then x else y
```

9.1.1 Возвращение результатов из модели

После того, как мы завершили все необходимые приготовления, мы теперь можем увидеть, как параметрическая модель может быть задана еще для более широкого использования метода Монте-Карло. Сам метод безусловно поддерживается. Речь идет о более полном использовании. Это уже ближе к планированию эксперимента.

Мы представляем каждый внешний параметр как вычисление `Parameter`. Чтобы сделать такой параметр воспроизводимым внутри каждого имитационного запуска, случайный параметр должен быть мемоизирован с помощью функции `memoParameter`. Тогда параметр будет возвращать одно и то же значение внутри запуска, но при следующем запуске будет, возможно, выбрано другое значение, которое может быть случайным.

Также эта модель возвращает `Results` внутри вычисления `Simulation`. Такие результаты могут быть обработаны затем или просто выведены на терминал.

```
{-# LANGUAGE RecursiveDo #-}

module Model
       (-- * Имитационная модель
        model,
        -- * Названия переменных
        netIncomeName,
        netCashFlowName,
        npvIncomeName,
        npvCashFlowName,
```

```
        -- * Внешние параметры
        Parameters(..),
        defaultParams,
        randomParams) where

import Control.Monad

import Simulation.Aivika
import Simulation.Aivika.SystemDynamics
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

-- | Параметры модели.
data Parameters =
  Parameters { paramsTaxDepreciationTime    :: Parameter Double,
               paramsTaxRate                :: Parameter Double,
               paramsAveragePayableDelay    :: Parameter Double,
               paramsBillingProcessingTime  :: Parameter Double,
               paramsBuildingTime           :: Parameter Double,
               paramsDebtFinancingFraction  :: Parameter Double,
               paramsDebtRetirementTime     :: Parameter Double,
               paramsDiscountRate           :: Parameter Double,
               paramsFractionalLossRate     :: Parameter Double,
               paramsInterestRate           :: Parameter Double,
               paramsPrice                  :: Parameter Double,
               paramsProductionCapacity     :: Parameter Double,
               paramsRequiredInvestment     :: Parameter Double,
               paramsVariableProductionCost :: Parameter Double }

-- | Параметры модели по умолчанию.
defaultParams :: Parameters
defaultParams =
  Parameters { paramsTaxDepreciationTime    = 10,
               paramsTaxRate                = 0.4,
               paramsAveragePayableDelay    = 0.09,
               paramsBillingProcessingTime  = 0.04,
               paramsBuildingTime           = 1,
               paramsDebtFinancingFraction  = 0.6,
               paramsDebtRetirementTime     = 3,
               paramsDiscountRate           = 0.12,
               paramsFractionalLossRate     = 0.06,
               paramsInterestRate           = 0.12,
               paramsPrice                  = 1,
               paramsProductionCapacity     = 2400,
               paramsRequiredInvestment     = 2000,
               paramsVariableProductionCost = 0.6 }
```

```
-- | Случайные параметры для метода Монте-Карло.
randomParams :: IO Parameters
randomParams =
  do averagePayableDelay    <- memoParameter $ randomUniform 0.07 0.11
     billingProcessingTime  <- memoParameter $ randomUniform 0.03 0.05
     buildingTime           <- memoParameter $ randomUniform 0.8 1.2
     fractionalLossRate     <- memoParameter $ randomUniform 0.05 0.08
     interestRate           <- memoParameter $ randomUniform 0.09 0.15
     price                  <- memoParameter $ randomUniform 0.9 1.2
     productionCapacity     <- memoParameter $ randomUniform 2200 2600
     requiredInvestment     <- memoParameter $ randomUniform 1800 2200
     variableProductionCost <- memoParameter $ randomUniform 0.5 0.7
     return defaultParams {
       paramsAveragePayableDelay    = averagePayableDelay,
       paramsBillingProcessingTime  = billingProcessingTime,
       paramsBuildingTime           = buildingTime,
       paramsFractionalLossRate     = fractionalLossRate,
       paramsInterestRate           = interestRate,
       paramsPrice                  = price,
       paramsProductionCapacity     = productionCapacity,
       paramsRequiredInvestment     = requiredInvestment,
       paramsVariableProductionCost =
         variableProductionCost }

-- | Это сама модель, которая возвращает результаты эксперимента.
model :: Parameters -> Simulation Results
model params =
  mdo let getParameter f = liftParameter $ f params

      -- уравнения ниже даны в произвольном порядке!

      bookValue <- integ (newInvestment - taxDepreciation) 0
      let taxDepreciation = bookValue / taxDepreciationTime
          taxableIncome = grossIncome - directCosts - losses
                          - interestPayments - taxDepreciation
          production = availableCapacity
          availableCapacity = ifDynamics (time .>=. buildingTime)
                              productionCapacity 0
          taxDepreciationTime = getParameter paramsTaxDepreciationTime
          taxRate = getParameter paramsTaxRate
      accountsReceivable <- integ (billings - cashReceipts - losses)
                            (billings / (1 / averagePayableDelay
                                         + fractionalLossRate))
      let averagePayableDelay = getParameter paramsAveragePayableDelay
      awaitingBilling <- integ (price * production - billings)
```

```
                    (price * production * billingProcessingTime)
      let billingProcessingTime =
            getParameter paramsBillingProcessingTime
          billings = awaitingBilling / billingProcessingTime
          borrowing = newInvestment * debtFinancingFraction
          buildingTime = getParameter paramsBuildingTime
          cashReceipts = accountsReceivable / averagePayableDelay
      debt <- integ (borrowing - principalRepayment) 0
      let debtFinancingFraction =
            getParameter paramsDebtFinancingFraction
          debtRetirementTime = getParameter paramsDebtRetirementTime
          directCosts = production * variableProductionCost
          discountRate = getParameter paramsDiscountRate
          fractionalLossRate = getParameter paramsFractionalLossRate
          grossIncome = billings
          interestPayments = debt * interestRate
          interestRate = getParameter paramsInterestRate
          losses = accountsReceivable * fractionalLossRate
          netCashFlow = cashReceipts + borrowing - newInvestment
                        - directCosts - interestPayments
                        - principalRepayment - taxes
          netIncome = taxableIncome - taxes
          newInvestment = ifDynamics (time .>=. buildingTime)
                          0 (requiredInvestment / buildingTime)
      npvCashFlow <- npv netCashFlow discountRate 0 1
      npvIncome <- npv netIncome discountRate 0 1
      let price = getParameter paramsPrice
          principalRepayment = debt / debtRetirementTime
          productionCapacity = getParameter paramsProductionCapacity
          requiredInvestment = getParameter paramsRequiredInvestment
          taxes = taxableIncome * taxRate
          variableProductionCost =
            getParameter paramsVariableProductionCost

      return $
        results
        [resultSource netIncomeName "Net income" netIncome,
         resultSource netCashFlowName "Net cash flow" netCashFlow,
         resultSource npvIncomeName "NPV income" npvIncome,
         resultSource npvCashFlowName "NPV cash flow" npvCashFlow]

-- Названия переменных, которые нам интересны.
netIncomeName   = "netIncome"
netCashFlowName = "netCashFlow"
npvIncomeName   = "npvIncome"
npvCashFlowName = "npvCashFlow"
```

Теперь мы можем применить метод Монте-Карло к этой параметрической модели, например, для того, чтобы определить насколько чувствительны некоторые из переменных к случайным внешним параметрам.

Основная идея заключается в том, что не только системы ОДУ могут быть параметрическими. Нет никакой разницы, интегрируем ли мы численно, запускаем ли дискретно-событийную модель или запускаем имитацию агентов. Внешние параметры — это просто вычисления `Parameter`, которые могут быть использованы в других имитационных вычислениях.

9.1.2 Задание экспериментов

В отличие от других примеров здесь мы, в действительности, определяем сразу два эксперимента: один для имитации по методу Монте-Карло, а другой для более простого единичного запуска модели.

```
module Experiment (monteCarloExperiment, singleExperiment,
                   monteCarloGenerators, singleGenerators) where

import Control.Monad

import Data.Monoid

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

import Model

-- | Параметры моделирования.
specs = Specs 0 5 0.015625 RungeKutta4 SimpleGenerator

-- | Эксперимент для метода Монте-Карло.
monteCarloExperiment :: Experiment
monteCarloExperiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1000,
    experimentTitle = "Financial Model (the Monte-Carlo simulation)",
    experimentDescription =
      "Financial Model (the Monte-Carlo simulation) as described in " ++
      "Vensim 5 Modeling Guide, Chapter Financial Modeling and Risk." }

netIncome = resultByName netIncomeName
```

```
npvIncome = resultByName npvIncomeName

netCashFlow = resultByName netCashFlowName
npvCashFlow = resultByName npvCashFlowName

monteCarloGenerators :: ChartRendering r => [WebPageGenerator r]
monteCarloGenerators =
  [outputView defaultExperimentSpecsView,
   outputView defaultInfoView,
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "Chart 1",
     deviationChartPlotTitle =
       "The deviation chart for Net Income and Cash Flow",
     deviationChartLeftYSeries = netIncome <> netCashFlow },
   outputView $ defaultDeviationChartView {
     deviationChartTitle = "Chart 2",
     deviationChartPlotTitle =
       "The deviation chart for Net Present Value of Income " ++
       "and Cash Flow",
     deviationChartLeftYSeries = npvIncome <> npvCashFlow },
   outputView $ defaultFinalHistogramView {
     finalHistogramTitle = "Histogram 1",
     finalHistogramPlotTitle ="Histogram for Net Income and Cash Flow",
     finalHistogramSeries = netIncome <> netCashFlow },
   outputView $ defaultFinalHistogramView {
     finalHistogramTitle = "Histogram 2",
     finalHistogramPlotTitle =
       "Histogram for Net Present Value of Income and Cash Flow",
     finalHistogramSeries = npvIncome <> npvCashFlow },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "Summary 1",
     finalStatsSeries = netIncome <> netCashFlow },
   outputView $ defaultFinalStatsView {
     finalStatsTitle = "Summary 2",
     finalStatsSeries = npvIncome <> npvCashFlow } ]

-- | Эксперимент с единичным запуском имитации.
singleExperiment :: Experiment
singleExperiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentTitle = "Financial Model",
    experimentDescription =
      "Financial Model as described in " ++
      "Vensim 5 Modeling Guide, Chapter Financial Modeling and Risk." }
```

```
singleGenerators :: ChartRendering r => [WebPageGenerator r]
singleGenerators =
  [outputView defaultExperimentSpecsView,
   outputView defaultInfoView,
   outputView $ defaultTimeSeriesView {
     timeSeriesTitle = "Time Series 1",
     timeSeriesPlotTitle = "Time series of Net Income and Cash Flow",
     timeSeriesLeftYSeries = netIncome <> netCashFlow },
   outputView $ defaultTimeSeriesView {
     timeSeriesTitle = "Time Series 2",
     timeSeriesPlotTitle =
       "Time series of Net Present Value for Income and Cash Flow",
     timeSeriesLeftYSeries = npvIncome <> npvCashFlow },
   outputView $ defaultTableView {
     tableTitle = "Table",
     tableSeries = netIncome <> netCashFlow <>
       npvIncome <> npvCashFlow } ]
```

9.1.3 Вывод графиков

Нам понадобятся два разных вызова имитационных экспериментов, чтобы запустить наши эксперименты друг за другом.

Интерфейс графиков на основе Cairo

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Cairo

import Graphics.Rendering.Chart.Backend.Cairo

import Model
import Experiment

main = do

  -- запустить обычную имитацию
  putStrLn "*** The simulation with default parameters..."
  runExperiment
    singleExperiment singleGenerators
    (WebPageRenderer (CairoRenderer PNG) experimentFilePath)
    (model defaultParams)
  putStrLn ""
```

```
  -- запустить эксперимент Монте-Карло
  putStrLn "*** The Monte-Carlo simulation..."
  randomParams >>= runExperimentParallel
    monteCarloExperiment monteCarloGenerators
    (WebPageRenderer (CairoRenderer PNG) experimentFilePath) . model
```

Интерфейс графиков на основе Diagrams

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Diagrams

import Graphics.Rendering.Chart.Backend.Diagrams

import Model
import Experiment

main = do
  fonts <- loadCommonFonts
  let renderer = DiagramsRenderer SVG (return fonts)

  -- запустить обычную имитацию
  putStrLn "*** The simulation with default parameters..."
  runExperiment
    singleExperiment singleGenerators
    (WebPageRenderer renderer experimentFilePath) (model defaultParams)
  putStrLn ""

  -- запустить эксперимент Монте-Карло
  putStrLn "*** The Monte-Carlo simulation..."
  randomParams >>= runExperimentParallel
    monteCarloExperiment monteCarloGenerators
    (WebPageRenderer renderer experimentFilePath) . model
```

9.1.4 Запуск имитационных экспериментов

При использовании интерфейса графиков на основе Diagrams оба имитационных эксперимента, запущенных друг за другом, где первый эксперимент содержал единичный прогон, а последний содержал 1000 прогонов, длились в течение 34 секунд на моем ноутбуке. При использовании интерфейса на основе Cairo оба эксперимента завершились через 23 секунды.

Вы можете увидеть один из графиков на рисунке 9.1.

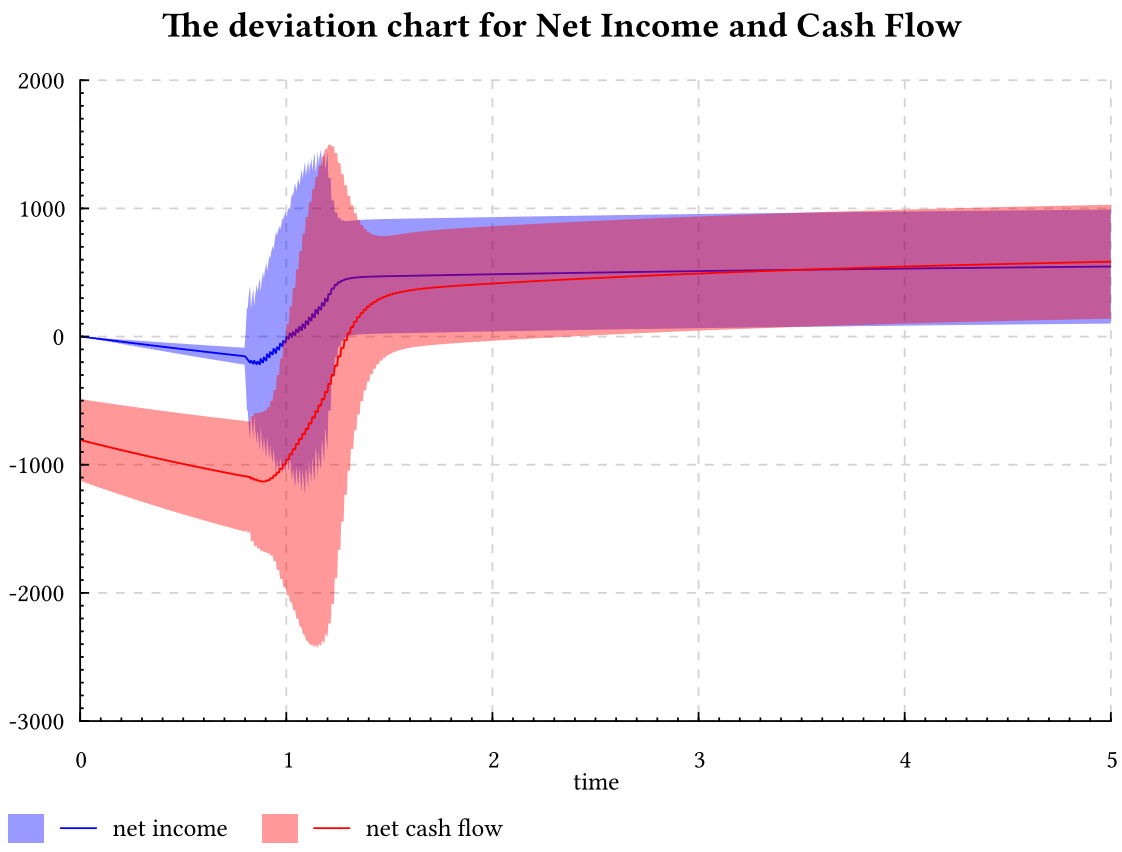


Рис. 9.1: График отклонения для чистого дохода и денежного потока.

9.2 Пример: использование массивов

Некоторые продавцы предлагают разные версии своих программных продуктов по имитационному моделированию, где одним из главных преимуществ использования коммерческой версии называют возможность задействования массивов.

В Айвике не требуется особая поддержка для массивов. Массивы могут быть естественным образом использованы в рамках моделирующих вычислений.

9.2.1 Возвращение результатов из модели

Давайте возьмем модель Linear Array из Berkeley Madonna[7], чтобы продемонстрировать основную идею.

```
{-# LANGUAGE RecursiveDo #-}

module Model (model) where

import Data.Array
import Control.Monad
import Control.Monad.Trans

import qualified Data.Vector as V

import Simulation.Aivika
import Simulation.Aivika.SystemDynamics
import Simulation.Aivika.Experiment

-- | Это аналог функции 'V.generateM'.
generateArray :: (Ix i, Monad m) => (i, i) -> (i -> m a) -> m (Array i a)
generateArray bnds generator =
  do ps <- forM (range bnds) $ \i ->
       do x <- generator i
          return (i, x)
     return $ array bnds ps

model :: Int -> Simulation Results
model n =
  mdo m <- generateArray (1, n) $ \i ->
        integ (q + k * (c!(i - 1) - c!i) + k * (c!(i + 1) - c!i)) 0
      let c =
            array (0, n + 1) [(i, if (i == 0) || (i == n + 1)
```

```
                                then 0
                                else (m!i / v)) | i <- [0 .. n + 1]]
          q = 1
          k = 2
          v = 0.75
      return $ results
        [resultSource "t" "time" time,
         resultSource "m" "M" m,
         resultSource "c" "C" c]
```

Приведенный выше код использует стандартный модуль `Array`. Если бы мы использовали модуль `Vector`, то тогда нам вовсе не нужна была бы функция `generateArray`, которую можно было бы заменить на библиотечную функцию `generateM` из модуля `Vector`.

Наша модель создает массив интегралов. Похожим образом мы могли бы использовать массивы в дискретно-событийной или агентной модели.

9.2.2 Задание эксперимента

В эксперименте мы хотели бы взглянуть на массивы с разных ракурсов.

```
module Experiment (experiment, generators) where

import Data.Monoid

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

specs = Specs { spcStartTime = 0,
                spcStopTime = 500,
                spcDT = 0.1,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 1,
    experimentTitle = "Linear Array",
    experimentDescription =
      "Model Linear Array as described in " ++
      "the examples included in Berkeley-Madonna." }
```

```
t = resultByName "t"
m = resultByName "m"
c = resultByName "c"

generators :: ChartRendering r => [WebPageGenerator r]
generators =
  [outputView defaultExperimentSpecsView,
   outputView $ defaultTableView {
     tableSeries = t <> m <> c },
   outputView $ defaultTimeSeriesView {
     timeSeriesLeftYSeries = m,
     timeSeriesWidth = 800,
     timeSeriesHeight = 800 },
   outputView $ defaultTimeSeriesView {
     timeSeriesRightYSeries = c,
     timeSeriesWidth = 800,
     timeSeriesHeight = 800 },
   outputView $ defaultTimeSeriesView {
     timeSeriesLeftYSeries = m,
     timeSeriesRightYSeries = c,
     timeSeriesWidth = 800,
     timeSeriesHeight = 800 },
   outputView $ defaultXYChartView {
     xyChartXSeries = t,
     xyChartLeftYSeries = m,
     xyChartWidth = 800,
     xyChartHeight = 800 },
   outputView $ defaultXYChartView {
     xyChartXSeries = t,
     xyChartRightYSeries = c,
     xyChartWidth = 800,
     xyChartHeight = 800 },
   outputView $ defaultXYChartView {
     xyChartXSeries = t,
     xyChartLeftYSeries = m,
     xyChartRightYSeries = c,
     xyChartWidth = 800,
     xyChartHeight = 800 } ]
```

9.2.3 Вывод графиков

Поскольку наша модель зависит от числового параметра, то мы определяем другие исполняемые программы для запуска, хотя они и очень похожи на те,

что мы использовали ранее.

Интерфейс на основе Cairo

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Cairo

import Graphics.Rendering.Chart.Backend.Cairo

import Model
import Experiment

main =
  runExperiment experiment generators
  (WebPageRenderer (CairoRenderer PNG) experimentFilePath)
  (model 51)
```

Интерфейс на основе Diagrams

```
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart
import Simulation.Aivika.Experiment.Chart.Backend.Diagrams

import Graphics.Rendering.Chart.Backend.Diagrams

import Model
import Experiment

main =
  do fonts <- loadCommonFonts
     let renderer = DiagramsRenderer SVG (return fonts)
     runExperiment experiment generators
       (WebPageRenderer renderer experimentFilePath)
       (model 51)
```

Здесь нет необходимости в параллелизме при запуске эксперимента, поскольку у нас только единичный прогон имитации.

9.2.4 Запуск имитационного эксперимента

Во время использования интерфейса графиков на основе Diagrams единичный прогон длился 39 секунд, тогда как он же длился 7 секунд при использовании интерфейса на основе Cairo. Такая большая разница связана с тем, что

графические файлы SVG содержат слишком много информации. Пожалуйста, будьте осторожны с этим подводным камнем!

Вы можете увидеть один из соответствующих графиков на рисунке 9.2.

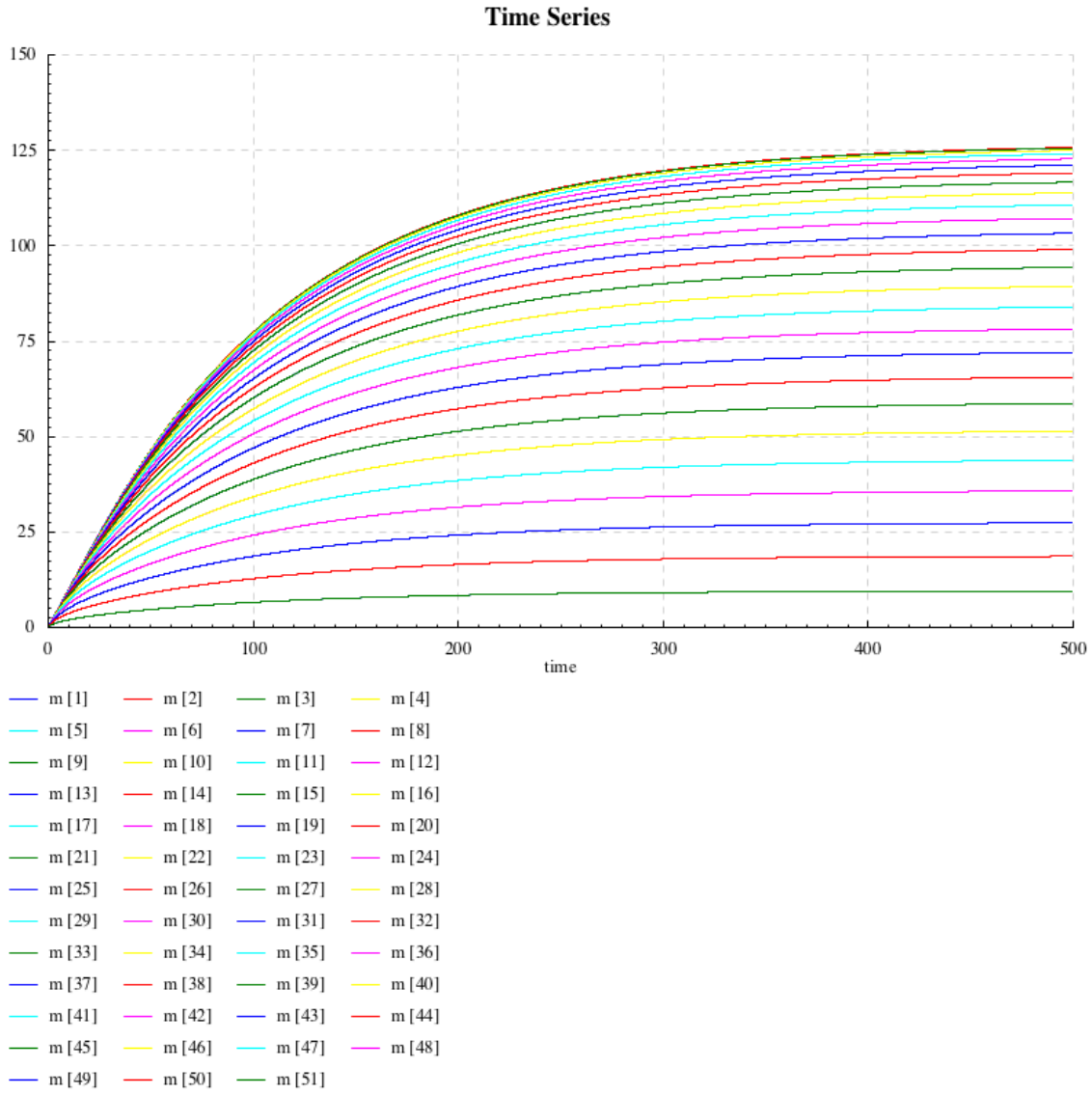


Рис. 9.2: Временные ряды для массивов.

Глава 10

DSL по аналогии с GPSS

Айвика поддерживает свой внутренний предметно-ориентированный язык (*англ.* DSL, domain specific language), который похож на популярный язык моделирования GPSS[14], только это по-прежнему тот же самый Haskell.

Обратите внимание на то, что GPSS-подобный DSL неэквивалентен оригинальному языку GPSS, хотя результаты моделирования в некоторых случаях могут быть очень похожи, но могут и отличаться в других случаях.

Пакет `aivika-gpss` реализует большинство блоков GPSS, но главная разница заключена в следующем.

Подобно GPSS этот пакет пытается трактовать приоритеты транзактов должным образом внутри каждого блока. Здесь он работает очень похожим образом даже для таких нетривиальных блоков как `PREEMPT`, `GATHER` и `ASSEMBLE`. Тем не менее, в отличие от GPSS блоки функционируют независимо друг от друга, причем, начиная только с версии 0.7, приоритеты транзактов начали влиять на порядок активации блоков, но при одинаковых приоритетах такой порядок по-прежнему непредсказуем.

10.1 Блоки и транзакты

Блоки и *генераторы* являются основными вычислениями в GPSS-подобном DSL. Они имеют следующее определение[1]:

```
data Block a b = Block { blockProcess :: a -> Process b }
```

[1]Читатель с опытом на языке Haskell мог бы заметить, что вычисление `Block`, на самом деле, является стрелкой Клейсли, но без `ArrowLoop`, поскольку `Process` не является `MonadFix`.

```
newtype GeneratorBlock a =
  GeneratorBlock { runGeneratorBlock :: Block a () -> Process () }
```

Это следующее развитие идеи использования вычисления `Process` как строительного элемента для композиции. Здесь напомним, что `Process` означает дискретный процесс. В общем-то, вычисления `Block` и `Process` эквивалентны[2].

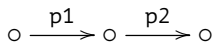
Здесь вычисление `Block` соответствует блоку GPSS, который имеет вход типа `a` и выход типа `b`. Вычисление `GeneratorBlock` соответствует конструкции `GENERATE` из GPSS, которое, будучи примененным к некоторому терминальному блоку `Block`, возвращает действие, которое в свою очередь моделирует прохождение порождаемых элементов через соответствующий блок. Блок называется *терминальным*, если его выход, то есть второй параметр типа, является пустым кортежем как в следующем примере: `Block a ()`.

Тогда конструкции `TERMINATE` из GPSS будет соответствовать следующее вычисление:

```
terminateBlock :: Block a ()
```

Поскольку `Block` является представителем `Category`, то мы можем соединять блоки в цепочку:

```
p1 :: Block a b
p2 :: Block b c

p3 :: Block a c
p3 = p1 >>> p2
```



Для создания генераторов мы можем использовать вычисление `Stream`, особенно его случайные потоки.

```
streamGeneratorBlock :: Stream (Arrival a)
                        -> Int
                        -> GeneratorBlock (Transact a)

streamGeneratorBlock0 :: Stream (Arrival a)
                         -> GeneratorBlock (Transact a)
```

[2]Это утверждение в одну сторону очевидно, а в обратную сторону для преобразования из `Block` в `Process` доказательство может быть непростым.

Здесь первая функция принимает входной поток и некоторый целый приоритет, который будет назначен новым транзактам. Вторая функция назначает нулевой приоритет. Обе функции возвращают по генератору транзактов.

Каждый транзакт имеет целочисленный приоритет, а также транзакт несет в себе некоторое значение произвольного типа.

```
data Transact a

transactPriority :: Transact a -> Int
transactValue :: Transact a -> a
```

По мере прохождения транзакта через цепочку блоков мы можем менять как приоритет, так и само значение, которое несет в себе транзакт, поскольку это функтор.

```
instance Functor Transact
```

Очень важно отметить, что вычисление `Block` может обрабатывать транзакты параллельно в отличие от вычисления `Stream`.

В Айвике большинство моделирующих блоков GPSS имеют непосредственные или очень близкие аналоги. В других случаях блоки могут быть вручную записаны как вычисления `Block`, например в случае блока `SELECT`, который требует информации об очередях и других сущностях, к которым этот блок присоединен.

Таблица 10.1: Соответствие блокам GPSS.

Конструкция GPSS	Аналог в Айвике
`GENERATE`	`streamGeneratorBlock` / `signalGeneratorBlock`
`TERMINATE`	`terminateBlock`
цепочка блоков	композиция вычислений
`ADVANCE`	`advanceBlock`
`ASSEMBLE`	`assembleBlock`
`ASSIGN`	`assignBlock`
`DEPART`	`departBlock`
`ENTER`	`enterBlock`
`GATHER`	`gatherBlock`
`LEAVE`	`leaveBlock`
`LINK`	`linkBlock`

LOOP	loopBlock
MATCH	matchBlock
PREEMPT	preemptBlock
PRIORITY	priorityBlock
QUEUE	queueBlock
RELEASE	releaseBlock
RETURN	returnBlock
SEIZE	seizeBlock
SPLIT	splitBlock
TEST	awaitingTestBlock / transferringTestBlock
UNLINK	unlinkBlock

Идея заключена в том, что мы естественным образом можем комбинировать другие вычисления в рамках вычисления `Block`. Например это буквально означает, что мы можем использовать агенты внутри имитационной модели GPSS, и наоборот.

10.2 Пример: использование GPSS

Ниже рассмотрена имитационная модель[14], которая демонстрирует использование GPSS-подобного DSL.

> Профессор дает консультации. Студенты приходят на консультацию к профессору в его приемные часы, подчиняясь дисциплине "первым пришел — первым обслужен". Если звонит телефон, то звонок прерывает разговор профессора со студентом. Профессор возобновляет беседу со студентом только после окончания разговора по телефону. С этим прерыванием связана некоторая задержка по времени. Когда вновь возобновляется беседа со студентом, некоторое время неизбежно уходит на то, чтобы восстановить тот момент, с которого разговор был прерван. Этот временной излишек необходимо добавить к величине времени пребывания студента у профессора.
>
> Сделаны следующие предположения.
>
> - Единицей времени в модели является 0,01 мин.

- Телефон звонит каждые 20 +/- 5 минут.
- Длительность телефонного разговора распределена экспоненциально со средним 2 мин.
- Длительность разговора со студентом также распределена экспоненциально со средним 10 мин.
- При прерывании беседы возникает минутный излишек во времени, равный 3 мин.

Модель GPSS приведена ниже. Для простоты мы полагаем, что всего 20 студентов ожидают консультации.

```
        GENERATE 2000,500,,,1
        GATE NI PROF,Busy
        PREEMPT PROF,PR,Add,5
        ADVANCE (Exponential(1,0,200))
        RETURN PROF
Busy    TERMINATE

        GENERATE ,,,20
        QUEUE LINE
        SEIZE PROF
        DEPART LINE
        ADVANCE (Exponential(1,0,1000))
LetGo   RELEASE PROF
        TERMINATE

Add     ASSIGN 5+,300
        ADVANCE P5
        TRANSFER ,LetGo

        GENERATE 20000
        TERMINATE 1

        START 1
```

Здесь мы видим блок вытеснения прибора, что моделирует телефонный звонок. Это существенно усложняет модель. К счастью, в Айвике есть непосредственный аналог.

10.2.1 Возвращение результатов из модели

Наша модель очень похожа, только она написана на чистом языке Haskell. Также мы сбрасываем статистику в момент времени 4000.

```
module Model (model) where

import Prelude hiding (id)

import Control.Category
import Control.Monad.Trans

import Data.Maybe

import Simulation.Aivika
import Simulation.Aivika.GPSS
import qualified Simulation.Aivika.GPSS.Queue as Q

model :: Simulation Results
model =
  do line <- runEventInStartTime Q.newQueue
     prof <- runEventInStartTime newFacility

     let phoneCallStream = randomUniformStream (2000 - 500) (2000 + 500)
         studentStream   = takeStream 20 $ randomUniformStream 0 0

     let phoneCalls     = streamGeneratorBlock phoneCallStream 1
         phoneCallChain =
           Block (\a ->
                   do f <- liftEvent (facilityInterrupted prof)
                      if f
                        then blockProcess (transferBlock busy) a
                        else return a) >>>
           preemptBlock prof
           (PreemptBlockMode { preemptBlockPriorityMode = True,
                               preemptBlockTransfer     = Just add,
                               -- preemptBlockTransfer     = Nothing,
                               preemptBlockRemoveMode   = False }) >>>
           advanceBlock (randomExponentialProcess_ 200) >>>
           returnBlock prof >>>
           busy
         busy           = terminateBlock

         students       = streamGeneratorBlock studentStream 0
         studentChain   =
           queueBlock line 1 >>>
           seizeBlock prof >>>
           departBlock line 1 >>>
           advanceBlock (randomExponentialProcess_ 1000) >>>
           letGo
         letGo          =
```

```
          releaseBlock prof >>>
          terminateBlock
        add dt0        =
          let dt = maybe 0 id dt0
          in advanceBlock (holdProcess (dt + 300)) >>>
             transferBlock letGo

    runProcessInStartTime $
      runGeneratorBlock phoneCalls phoneCallChain

    runProcessInStartTime $
      runGeneratorBlock students studentChain

    runEventInStartTime $
      enqueueEvent 4000 $
      do Q.resetQueue line
         resetFacility prof

    return $
      results
      [resultSource "line" "Line" line,
       resultSource "prof" "Prof" prof]
```

10.2.2 Задание эксперимента

В рамках эксперимента мы хотели бы увидеть некоторую статистику, например, как будет уменьшаться размер очереди. Соответствующий временной ряд назовем `lineContent`.

```
{-# LANGUAGE FlexibleContexts #-}

module Experiment (experiment, generators) where

import Data.Monoid

import Control.Arrow

import Simulation.Aivika
import Simulation.Aivika.Experiment
import Simulation.Aivika.Experiment.Chart

import qualified Simulation.Aivika.Results.Transform as T
import qualified Simulation.Aivika.GPSS.Results.Transform as GpssT

-- | Параметры моделирования.
```

```
specs = Specs { spcStartTime = 0.0,
                spcStopTime = 20000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

-- | Имитационный эксперимент.
experiment :: Experiment
experiment =
  defaultExperiment {
    experimentSpecs = specs,
    experimentRunCount = 10000,
    -- experimentRunCount = 100,
    experimentTitle = "The GPSS module 7.31" }

line = GpssT.Queue $ resultByName "line"
lineContent = GpssT.queueContent line
lineContentStats = T.tr $ GpssT.queueContentStats line
lineWaitTime = T.tr $ GpssT.queueWaitTime line
lineNonZeroEntryWaitTime = T.tr $ GpssT.queueNonZeroEntryWaitTime line

prof = GpssT.Facility $ resultByName "prof"
profUtilCount = GpssT.facilityUtilisationCount prof
profUtilCountStats = T.tr $ GpssT.facilityUtilisationCountStats prof
profHoldingTime = T.tr $ GpssT.facilityHoldingTime prof

statsView title series =
  defaultFinalStatsView {
    finalStatsTitle = title,
    finalStatsSeries = series
  }

chartView title series =
  defaultDeviationChartView {
    deviationChartTitle = title,
    deviationChartRightYSeries = series
  }

histogramView title series =
  defaultFinalHistogramView {
    finalHistogramTitle = title,
    finalHistogramSeries = series
  }

generators :: ChartRendering r => [WebPageGenerator r]
generators =
```

```
[outputView defaultExperimentSpecsView,
 outputView defaultInfoView,
 outputView $ statsView "PROF Utilisation" profUtilCount,
 outputView $ chartView "PROF Utilisation" profUtilCount,
 outputView $ statsView "PROF Holding Time" profHoldingTime,
 outputView $ chartView "PROF Holding Time" profHoldingTime,
 outputView $ statsView "LINE Content" lineContent,
 outputView $ chartView "LINE Content" lineContent,
 outputView $ histogramView "LINE Content" lineContent,
 outputView $ statsView "LINE Wait Time" $
   lineWaitTime <>
   lineNonZeroEntryWaitTime,
 outputView $ chartView "LINE Wait Time" $
   lineWaitTime <>
   lineNonZeroEntryWaitTime]
```

10.2.3 Вывод графиков

Для запуска имитации мы можем выбрать один из интерфейсов для графиков. Код совершенно такой же, каким он был в разделе 1.4.3.

10.2.4 Запуск имитационного эксперимента

При использовании интерфейса графиков на основе Diagrams весь имитационный эксперимент с 10000 (десятью тысячами) прогонов длился 2 минуты и 16 секунд на моем MacBook Pro. При использовании интерфейса на основе Cairo тот же самый вычислительный эксперимент длился 1 минуту и 56 секунд.

Вы можете увидеть график отклонения для числа ожидающих консультации студентов на рисунке 10.1.

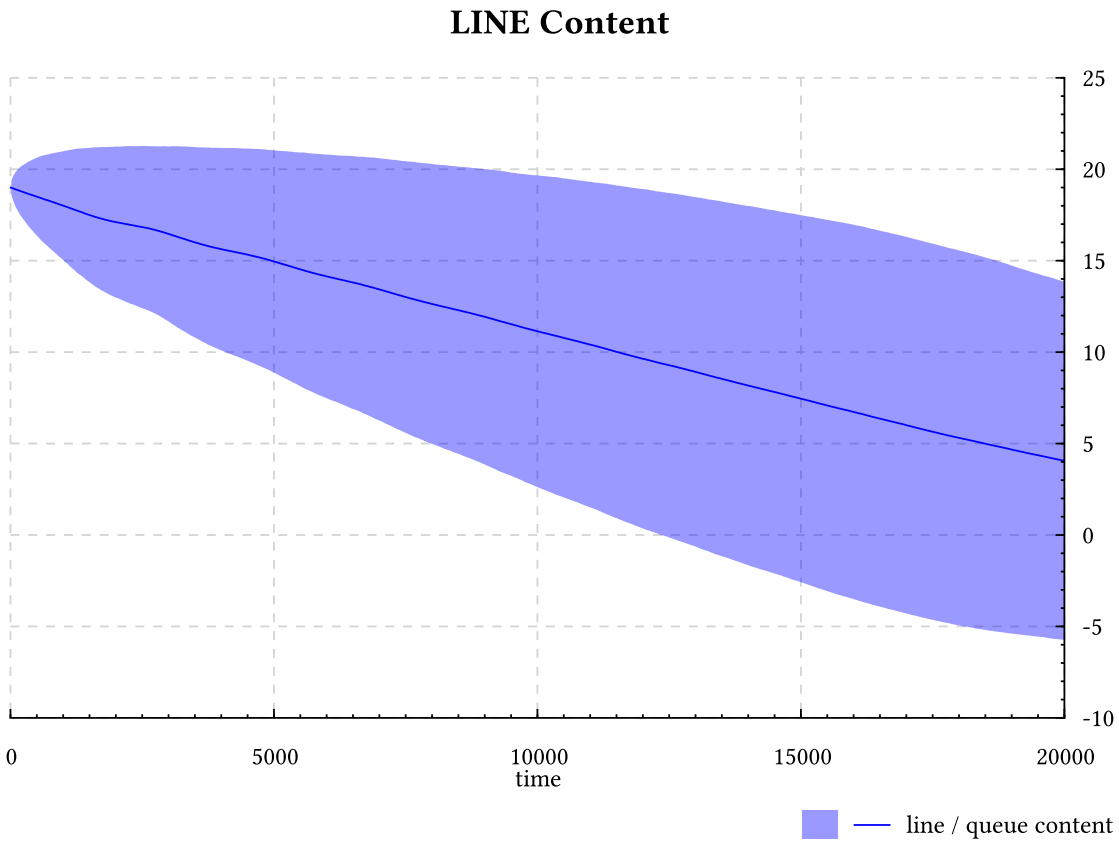


Рис. 10.1: Число ожидающих консультации студентов.

Часть II

Параллельное и распределенное моделирование

В этой части рассматривается параллельное и распределенное моделирования на основе оптимистичного метода деформации времени (*англ.* Time Warp). Это может быть полезно, если вы собираетесь использовать современные компьютеры с большим количеством вычислительных ядер или кластеры из таких компьютеров. Есть даже возможность соединить компьютеры, расположенные удаленно и разделенные огромными расстояниями, для того, чтобы создать распределенную имитацию, например, где узлы кластеров могут быть расположены в разных компьютерных центрах, соединенных через интернет.

Для этого Айвика поддерживает режим, который позволяет восстанавливать распределенную имитацию после временных ошибок соединения, которые неизбежны в сложных вычислительных сетях, но это работает в заданных пределах по времени ожидания, после истечения которого распределенная имитация автоматически останавливается. Это позволяет создавать вычислительные сервисы с определенными характеристиками по отклику.

Более того, есть задачи, которые не могут уместиться в памяти одного компьютера. Такие задачи могут быть решены только через создание распределенной имитационной модели.

Глава 11

Обобщение вычислений для имитации

Все рассмотренные ранее моделирующие вычисления могут быть обобщены для того, чтобы их можно было использовать в других видах моделирования, в частности, в распределенной имитации. Этот подход оказался настолько общим, что он прекрасно работает и для вложенного моделирования тоже.

11.1 Две версии библиотеки моделирования

На самом деле, существуют две версии основной библиотеки моделирования в Айвике. Есть первая, которая оптимизирована под последовательное моделирование. В дистрибутиве это пакет `aivika`. Также есть обобщение этой версии, которое может быть адаптировано к другим видам моделирования. Это уже пакет `aivika-transformers`.

Пакет `aivika` мог бы быть частным случаем `aivika-transformers`, но он оставлен по разным причинам. Во-первых, `aivika` имеет более простую документацию, что очень важно. Также он немного быстрее, хотя разница в скорости может в будущем уменьшиться. Также `aivika` покрывает основной и главный вариант использования, с которым моделисты сталкиваются в своей практике. Так что, я решил сохранить пакет `aivika`, хотя он мог бы быть полностью заменен более общим пакетом `aivika-transformers`.

Ранее в книге мы использовали пакет `aivika`, а вот текст далее, главным образом, относится к пакету `aivika-transformers`.

11.2 Замена IO абстрактным вычислением

Если мы повторим вычисление `Event` из первой части, посвященной последовательному моделированию, то мы тогда увидим, что оно возвращает некоторое значение в монаде `IO`.

```
newtype Event a = Event (Point -> IO a)
```

Обобщенная версия содержит другое вычисление с тем же названием `Event`, но параметризованное по некоторому другому вычислению, которому соответствует переменный тип `m`.

```
newtype Event m a = Event (Point -> m a)
```

Это обобщенное вычисление наряду с другими такими же часто является или монадическим трансформером, или очень близко к трансформеру. Отсюда и название соответствующего пакета `aivika-transformers`.

Для того, чтобы разделять разные вычисления с теми же названиями, обобщенная версия расположена в другом пространстве имен:

```
module Simulation.Aivika.Trans
```

Это далеко не самый идиоматический подход в языке Haskell, но он работает хорошо, так как позволяет легко конвертировать последовательные модели в распределенные и вложенные, что может быть важно, если вы собираетесь разрабатывать распределенную модель шаг за шагом, начиная с последовательного прототипа, который бы затем был расширен до распределенной модели.

Поэтому не только вычисления сохранили свои исходные названия, но также и соответствующие функции сохранили почти такие же сигнатуры — только теперь они обобщенные, где добавлены ограничения по классам типов.

```
enqueueEvent :: EventQueueing m => Double -> Event m () -> Event m ()
```

Класс типов `EventQueueing` буквально означает, что его представитель реализует очередь событий. Также существуют разные классы типов с одним названием `Ref`, чьи представители реализуют изменяемые ссылки с соответствующими гарантиями. Кроме этого, есть класс типов `MonadException`, представитель которого должен реализовать обработку ошибок вычисления `IO`.

Эти три ограничения составляют существо класса типов `MonadDES`: (1) возможность добавлять события; (2) возможность создавать и изменять ссылки и (3) возможность обрабатывать ошибки `IO`. Наконец, (4) представитель класса типов должен быть монадой.

```
class MonadDES m
```

Далее мы будем полагать, что наши такие моделирующие вычисления как `Event`, `Process` и другие параметризованы некоторым вычислением `MonadDES`.

Существует также похожий класс типов `MonadSD`, который позволяет интегрировать дифференциальные уравнения, но он редко используется.

```
class MonadSD m
```

Важно, что стандартная императивная монада `IO` является представителем обоих этих классов, что на самом деле означает то, что пакет `aivika` мог бы быть заменен его обобщающей теской `aivika-transformers`.

```
instance MonadDES IO
instance MonadSD IO
```

Только при использовании `IO` в обобщенных моделирующих вычислениях мы должны также импортировать модуль, где определены соответствующие реализации:

```
import Simulation.Aivika.IO
```

Существуют также другие соответствующие модули для распределенного моделирующего вычисления и вложенного моделирующего вычисления.

11.3 Обобщение последовательной модели

При условии, что у нас имеется последовательная имитационная модель, мы можем ее обобщить с той целью, чтобы ее можно было бы использовать для других видов моделирования, таких как распределенное или вложенное.

Обычно нам нужно будет импортировать другое пространство имен библиотек Айвики, заменив `Simulation.Aivika` на более общее пространство имен `Simulation.Aivika.Trans`. Также нам придется добавить параметрическое вычисление к моделирующим вычислениям.

Для примера мы можем ввести синоним типа, как показано ниже:

```
type DES = IO
```

Затем мы можем использовать в модели тип `Event DES a` вместо типа `Event a`.

Давайте возьмем нашу последовательную модель из раздела 2.6. Ее определение повторено ниже.

```
module Model(model) where

import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
meanRepairTime = 0.5

model :: Simulation Results
model =
  do totalUpTime <- newRef 0.0

     let machine :: Process ()
         machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)
              repairTime <-
                randomExponentialProcess meanRepairTime
              machine

     runProcessInStartTime machine
     runProcessInStartTime machine

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

     return $
       results
       [resultSource
        "upTimeProp"
        "The long-run proportion of up time (~ 0.66)"
        upTimeProp]
```

Теперь применив предложенную выше в этом разделе процедуру, мы можем обобщить последовательную имитационную модель.

```
module Model(model) where

import Control.Monad.Trans

import Simulation.Aivika.Trans
import Simulation.Aivika.IO

type DES = IO

meanUpTime = 1.0
meanRepairTime = 0.5

model :: Simulation DES (Results DES)
model =
  do totalUpTime <- newRef 0.0

     let machine :: Process DES ()
         machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)
              repairTime <-
                randomExponentialProcess meanRepairTime
              machine

     runProcessInStartTime machine
     runProcessInStartTime machine

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

     return $
       results
       [resultSource
        "upTimeProp"
        "The long-run proportion of up time (~ 0.66)"
        upTimeProp]
```

Смысл заключается в том, что мы можем параметризовать не только по вычислению `IO`. Похожим образом мы могли бы сконвертировать последо-

вательную модель в эквивалентную, используя вычисление для распределенного моделирования или вычисление для вложенного моделирования. Это важно, если мы желаем создать сложную модель из последовательной модели, взятой в качестве прототипа. Как пример, мы могли бы проверить и верифицировать прототип и только потом начать преобразовывать его в распределенную модель.

11.4 Написание обобщенного кода

Для того, чтобы писать обобщенный код, совсем не обязательно постоянно инстанцировать обобщенные вычисления по некоторому параметрическому вычислению, такому как `IO`. Так, если вы решите в своей функции добавить для типов ограничение `MonadDES` или `MonadSD`, то строго рекомендуется добавить также прагмы `INLINE` или `INLINABLE` по следующему примеру.

```
-- | Актуализировать обработчик событий в определенные точки времени.
enqueueEventWithTimes :: MonadDES m
                         => [Double]
                         -> Event m ()
                         -> Event m ()
{-# INLINABLE enqueueEventWithTimes #-}
enqueueEventWithTimes ts e = loop ts
  where loop []       = return ()
        loop (t : ts) = enqueueEvent t $ e >> loop ts
```

Без этой прагмы функция `enqueueEventWithTimes` была бы медленнее из-за ограничения `MonadDES`.

В то же самое время не было бы необходимости добавлять такую прагму, если ваша функция не имела бы ограничения по классу типов, например, если бы вы использовали уже параметризованное по конкретному типу моделирующее вычисление вида `Event IO a`.

Глава 12

Вычисление для распределенной имитации

Дистрибутив Айвики включает в себя пакет `aivika-distributed`, который позволяет нам создавать параллельные и распределенные имитационные модели[13] на основе оптимистичного метода деформации времени[3] (*англ.* Time Warp).

Мы запускаем *логические процессы*, возможно, что на разных компьютерах, или в разных потоках на том же компьютере, или в комбинации обоих подходов. Такие процессы посылают таким же процессам асинхронные *сообщения*, которые имеют временную метку. Временная метка — это модельное время, в которое заданное сообщение должно быть обработано адресатом.

Основная проблема распределенного моделирования следующая. Может случиться так называемый *парадокс времени*, когда некоторому логическому процессу приходит сообщение, которое бы смотрело на прошлое состояние этого процесса, где сообщение должно было бы быть обработанным еще в прошлом времени логического процесса.

Существуют *консервативные* методы, которые исключают саму возможность возникновения парадокса времени. Напротив, существуют также *оптимистичные* методы, которые допускают возникновение парадокса, но которые предоставляют средства для такого отката имитационной модели, чтобы проблемное сообщение могло бы быть обработано именно в тот момент модельного времени, на какое это сообщение назначено для обработки. Мы обращаем вспять модельное время логического процесса в случае необходимости. Этот откат может быть каскадным, охватывающим множество или даже все логические процессы при необходимости. Наиболее известный ме-

тод называется методом деформации времени, который Айвика и реализует.

Для передачи сообщений по сети Айвика использует Cloud Haskell, представленный пакетом `distributed-process`[1] и разными реализациями протокола, например `distributed-process-simplelocalnet`[2]. Возможно, что вам придется изучить то, как использовать эти пакеты, если вы собираетесь использовать распределенный модуль Айвики.

12.1 Вычисление DIO

Пакет `aivika-distributed` экспортирует модуль, который определяет вычисление `DIO` для распределенной имитации. Это вычисление позволяет создавать параллельные и распределенные имитационные модели.

```
module Simulation.Aivika.Distributed

data DIO a

instance MonadDES DIO
```

В распределенном модуле используется тип данных `ProcessId` из пакета `distributed-process`. В нашем случае он идентифицирует логический процесс. И не так важно, где находится этот процесс на самом деле. Он мог быть запущен в другом потоке того же самого процесса операционной системы, или мог быть запущен в другом процессе этого же компьютера, или этот идентификатор может представлять логический процесс, который был запущен на другом удаленном компьютере. Важно, что идентификатор `ProcessId` представляет прозрачно и единообразно некоторый логический процесс во всех этих случаях.

Для того, чтобы различать этот тип от идентификатора дискретного процесса, мы будем добавлять префикс `DP` к пространству имен, которое должно быть доступно через квалифицированный импорт пакета `distributed-process`.

```
import qualified Control.Distributed.Process as DP
import Control.Distributed.Process.Serializable
```

Существует следующая функция для того, чтобы послать сообщение другому логическому процессу.

[1] https://hackage.haskell.org/package/distributed-process

[2] https://hackage.haskell.org/package/distributed-process-simplelocalnet

```
sendMessage :: forall a. Serializable a
               => DP.ProcessId
               -> a
               -> Event DIO ()
```

Здесь класс типов `Serializable` обозначает нечто, что может быть сериализовано в двоичный поток данных, а затем послано по сети адресату. Пожалуйста, посмотрите документацию к пакету `distributed-process` для получения дополнительной информации.

Время получения сообщения будет равно времени отправки, когда вызывается функция `sendMessage`. Оригинальный метод деформации времени способен обработать откаты в таком случае. Однако Айвика поддерживает откаты дополнительного типа, которые возникают при попытке перезапустить вычисления, например, когда логический процесс временно переходит в недопустимое состояние, потому что не все сообщения еще были доставлены. Тогда мы не можем продолжить имитацию прямо сейчас, и мы должны дождаться прихода других сообщений. Эта ситуация описана в разделе 12.7.

Вот из-за этих откатов дополнительного вида строго рекомендуется вовсе не использовать функцию `sendMessage`, иначе распределенный модуль рискует впасть в бесконечный цикл. Время получения крайне желательно всегда делать больше времени отправки сообщения. Вы можете добавить всего лишь одну миллионную к текущему модельному времени, и это будет работать. Возможно, что в одной из будущих версий Айвики это ограничение будет снято.

Для того, чтобы задать точное время получения, в которое сообщение следует обработать другим логическим процессом, мы можем написать следующую функцию.

```
enqueueMessage ::
  forall a. Serializable a
  => DP.ProcessId
  -> Double
  -> a
  -> Event DIO ()
```

Каждый раз, как логический процесс получает некоторое сообщение, испускается соответствующий сигнал в назначенное время получения. Этот сигнал содержит само сообщение, которое может быть прочитано всеми слушателями сигнала.

```
messageReceived :: forall a. Serializable a => Signal DIO a
```

Если мы хотим, чтобы наш логический процесс получал все входящие сообщения заданного типа, то мы можем подписаться на получение такого сигнала, настроив соответствующую функцию на такой тип.

Только мы должны осторожно инициализировать логический процесс. Вам никогда не следует вызывать функцию `sendMessage` в начальный момент времени имитации. Если вы все же попытаетесь сделать это, то тогда есть высокая вероятность того, что другой логический процесс может получить соответствующее входящее сообщение и тут же потерять его еще до того, как тот логический процесс только попытается подписаться на обработку сигнала `messageReceived`. Время получения сообщения следует всегда делать больше начального времени моделирования.

12.2 Запуск вычисления DIO и сервера времени

Каждый запуск вычисления `DIO` равносилен запуску соответствующего логического процесса. Очевидно, что такие моделирующие вычисления как `Event DIO a` или `Process DIO a` могут быть в конечном итоге сведены к вычислению `DIO`. Вопрос заключается в том, как запустить само вычисление `DIO`?

Перед тем, как сделать это, мы должны ввести понятие так называемого *сервера времени*. По крайней мере, это то, как я назвал это в Айвике. Это название не обязано быть общепризнанным и распространенным в литературе.

Айвика использует алгоритм Самади[2, 13] для синхронизации *глобального виртуального времени* среди логических процессов распределенной имитации. Это нижняя оценка по всем значениям локального модельного времени каждого из логических процессов. Упомянутый сервер времени отвечает за выполнение такой синхронизации. Поэтому каждый запуск распределенной модели должен начаться запуском нового сервера времени, где сам сервер времени должен быть единственным и уникальным для всей распределенной имитации.

```
timeServer :: Int -> TimeServerParams -> DP.Process ()
```

Первый параметр задает кворум для логических процессов. Как только заданное число логических процессов подсоединится к серверу времени, этот сервер начнет синхронизацию глобального виртуального времени. Второй параметр задает параметры сервера времени. В наших примерах мы будем

использовать параметры по умолчанию, но вам следует ознакомиться с документацией `aivika-distributed` для получения дополнительной информации.

```
defaultTimeServerParams :: TimeServerParams
```

Поскольку сервер времени спроектирован так, чтобы он запускался на отдельном узле кластера, определена вспомогательная функция, которая упрощает такой вариант использования:

```
curryTimeServer :: (Int, TimeServerParams) -> DP.Process ()
```

Еще раз обратите внимание на то, что `DP.Process ()` — это не то же самое, что ранее обозначало `Process ()`. Первое означает вычисление, определенное в пакете `distributed-process`, тогда как последнее представляет собой некоторый дискретный процесс в рамках имитации.

Итак, после запуска единственного экземпляра сервера времени для всей распределенной модели мы получаем идентификатор `DP.ProcessId` для сервера времени и запускаем логические процессы один за другим: либо в разных локальных, либо в разных удаленных узлах кластера.

Здесь наиболее простая стратегия заключается в том, чтобы запустить один *ведущий* узел и множество *ведомых* узлов[3]. Перед запуском ведущего узла мы запускаем сначала ведомые узлы, возможно, на разных компьютерах. Только затем мы запускаем ведущий узел, который соединяется с ведомыми, а затем начинается новая распределенная имитация. После окончания имитации ведущий узел разрывает соединение с ведомыми узлами, а затем возвращает окончательный результат и останавливается, тогда как ведомые узлы могут продолжить свою работу, оставаясь на связи в ожидании новых подключений от другого нового ведущего узла, чтобы потом запустить новую распределенную имитацию. Пожалуйста, по этой теме ознакомьтесь с документацией `distributed-process-simplelocalnet`, чтобы лучше представить себе, как это работает.

Чтобы запустить новый логический процесс, мы должны передать идентификатор сервера времени в аргументах следующей функции.

```
runDIO :: DIO a -> DIOParams -> DP.ProcessId
          -> DP.Process (DP.ProcessId, DP.Process a)
```

[3] В английских названиях в примерах пока используется устаревшая терминология: master и slave, соответственно.

Первый параметр определяет соответствующее вычисление имитации, которое предварительно свели к вычислению `DIO`. Второй параметр задает параметры логического процесса, где мы будем для простоты использовать параметры по умолчанию, задаваемые значением `defaultDIOParams`.

```
defaultDIOParams :: DIOParams
```

Функция `runDIO` запускает вспомогательный процесс, который может принимать входящие сообщения от других логических процессов, а затем передавать такие сообщения соответствующему вычислению имитации. Тот вспомогательный процесс называется *входящим*. Идентификаторы входящего процесса и процесс моделирующего вычисления возвращаются функцией `runDIO`. Обратите внимание на то, что эта функция сама по себе не запускает соответствующее вычисление имитации. Мы должны его затем явно запустить. Такой подход достаточно гибок, и он позволяет нам запускать распределенные имитации в самых разных конфигурациях.

Для того, чтобы зарегистрировать себя в сервере времени, логический процесс должен применить функцию `registerDIO`. Если кворум по числу логических процессов будет достигнут, то тогда сервер времени начнет синхронизацию.

```
registerDIO :: DIO ()
```

После окончания имитации логическому процессу ведущего узла следует вызвать функцию `terminateDIO`, тогда как похожим логическим процессам ведомых узлов следует вызвать функцию `unregisterDIO`.

```
terminateDIO :: DIO ()
unregisterDIO :: DIO ()
```

Между прочим, мы могли бы запустить несколько распределенных имитаций на том же кластере. Тогда было бы такое же число экземпляров сервера времени, соответствующее числу запущенных имитаций. Разные логические процессы разделяли бы те же самые узлы кластера. Однако этот сценарий может быть непрактичным, поскольку такие имитации мешали бы друг другу, конкурируя за одни и те же ограниченные ресурсы компьютера или компьютеров.

12.3 Пример: имитация эквивалентная последовательной

Для того, чтобы показать, как распределенная имитация может быть запущена, мы возьмем ту же самую модель из раздела 11.3. Сама модель по-прежнему концептуально последовательная, но на самом деле она переписана как распределенная. Отличие от настоящей распределенной модели в том, что мы используем только один узел. Все остальное присуще уже распределенной имитации.

```
import Control.Monad
import Control.Monad.Trans
import Control.Concurrent
import qualified Control.Distributed.Process as DP
import Control.Distributed.Process.Node (initRemoteTable)
import Control.Distributed.Process.Backend.SimpleLocalnet

import Simulation.Aivika.Trans
import Simulation.Aivika.Distributed

meanUpTime = 1.0
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 10000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation DIO ()
model =
  do totalUpTime <- newRef 0.0

     let machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)
              repairTime <-
                randomExponentialProcess meanRepairTime
              machine

     runProcessInStartTime machine
     runProcessInStartTime machine
```

```
     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

     let rs =
           results
           [resultSource
            "upTimeProp"
            "The long-run proportion of up time (~ 0.66)"
            upTimeProp]

     printResultsInStopTime printResultSourceInEnglish rs

runModel :: DP.ProcessId -> DP.Process ()
runModel timeServerId =
  do DP.say "Started simulating..."
     let ps = defaultDIOParams { dioLoggingPriority = NOTICE }
         m =
           do registerDIO
              a <- runSimulation model specs
              terminateDIO
              return a
     (modelId, modelProcess) <- runDIO m ps timeServerId
     modelProcess

master = \backend nodes ->
  do liftIO . putStrLn $ "Slaves: " ++ show nodes
     let timeServerParams =
           defaultTimeServerParams { tsLoggingPriority = DEBUG }
     timeServerId  <-
       DP.spawnLocal $ timeServer 1 timeServerParams
     runModel timeServerId

main :: IO ()
main = do
  backend <- initializeBackend "localhost" "8080" rtable
  startMaster backend (master backend)
    where
      rtable :: DP.RemoteTable
      -- rtable = __remoteTable initRemoteTable
      rtable = initRemoteTable
```

Обратите внимание на ту церемонию, с какой мы запускаем ведущий узел. Это на самом деле является шаблоном для запуска логических процессов и

на ведомых узлах тоже. Мы далее будем его использовать. Пожалуйста, посмотрите документацию `distributed-process-simplelocalnet`, чтобы лучше привыкнуть к этому шаблону.

Также здесь мы добавили разные уровни логирования для сервера времени и логического процесса в соответствующих параметрах.

При запуске модели в WinGCHi на Windows 7, я получил следующий вывод[4], который слегка отредактировал для краткости, убрав метки времени и идентификаторы процессов.

```
GHCi, version 8.2.1: http://www.haskell.org/ghc/  :? for help
Prelude> :cd C:\Docs\Tests\test02-aivika-book
Prelude> :load "MachRep1Simple.hs"
[1 of 1] Compiling Main             ( MachRep1Simple.hs, interpreted )
Ok, 1 module loaded.
*Main> main
Slaves: []
[INFO] Time Server: starting...
Started simulating...
[DEBUG] Time Server: RegisterLogicalProcessMessage pid://localhost:808...
[INFO] Time Server: starting
[DEBUG] Time Server: computing the global time...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://localhost:8080:0:10 224.48...
[INFO] Time Server: providing the global time = Just 224.48176784549443
[DEBUG] Time Server: computing the global time...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://localhost:8080:0:10 10000.0
[INFO] Time Server: providing the global time = Just 10000.0
[DEBUG] Time Server: TerminateTimeServerMessage pid://localhost:8080...
[INFO] Time Server: start terminating...
[INFO] Time Server: terminate
[ERROR] Exception occurred: ProcessTerminationException
----------

-- simulation time
t = 10000.0

-- The long-run proportion of up time (~ 0.66)
upTimeProp = 0.6707069678156755
```

Если вы попытаетесь запустить модель в терминале macOS, то тогда неко-

[4] В современных версиях GHC придется явно указать использование некоторых пакетов. Поэтому будет предпочтительнее сейчас запускать как проект Cabal или Stack.

торые строки могут смешаться. Это связано с параллельным выводом информации в то же самое окно терминала.

Стоит заметить, что выброс исключения `ProcessTerminationException` на самом деле не является ошибкой. Он обозначает, что соответствующие входящий процесс или сервер времени закончили свою работу. Здесь мы могли бы увидеть от нуля до двух таких сообщений, созданных одним логическим процессом вместе с сервером времени. Просто так совпало, что мы увидели ровно одно сообщение, потому что ведущий процесс успел остановиться раньше. По устоявшейся традиции, распространенной в серверном программировании, соответствующее сообщение имеет уровень логирования `ERROR`. Во всех других случаях `ERROR` действительно бы означало ошибку, но не сейчас. Это было сделано намеренно для упрощения мониторинга за распределенной моделью.

12.4 Пример: делаем имитацию распределенной

В этом разделе будет показано, как модель из раздела 12.3 может быть преобразована в распределенную модель с передачей сообщений. Это достаточно искусственный пример. С точки зрения имитации, пример даже бесполезный, а сама итоговая модель много медленнее, чем исходная по очевидным причинам, но подобно многим книгам этот текст нацелен на описание инструментов, которые могли бы быть полезны для вас для того, чтобы вы могли создавать реальные модели. Итак, этот раздел посвящен описанию функций для передачи сообщений, а также посвящен тому, как вы можете запустить кластер.

Мы определяем в ведущем узле счетчик, который изменяем по получении сообщений от двух ведомых узлов. Каждый ведомый узел представляет собой отдельный станок. Поэтому ведущий узел должен подписаться на получение таких сообщений. Очевидно, что модель создает безумное количество бесполезных сообщений, но зато она показывает, как можно посылать и обрабатывать сообщения другим логическим процессом.

Поскольку здесь нет перезапуска вычислений, то мы все же можем для простоты изложения позволить себе использование функции `sendMessage`, хотя вам следует осознавать, что это рискованно. Пожалуйста, используйте функцию `enqueueMessage` с небольшим интервалом времени всегда, когда это только возможно.

```
{-# LANGUAGE TemplateHaskell #-}
```

```
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DeriveDataTypeable #-}

import System.Environment (getArgs)

import Data.Typeable
import Data.Binary

import GHC.Generics

import Control.Monad
import Control.Monad.Trans
import Control.Concurrent
import qualified Control.Distributed.Process as DP
import Control.Distributed.Process.Closure
import Control.Distributed.Process.Node (initRemoteTable)
import Control.Distributed.Process.Backend.SimpleLocalnet

import Simulation.Aivika.Trans
import Simulation.Aivika.Distributed

meanUpTime = 1.0
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 10000.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

newtype TotalUpTimeChange =
  TotalUpTimeChange { runTotalUpTimeChange :: Double }
  deriving (Eq, Ord, Show, Typeable, Generic)

instance Binary TotalUpTimeChange

-- | Ведомая подмодель.
slaveModel :: DP.ProcessId -> Simulation DIO ()
slaveModel masterId =
  do let machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                sendMessage masterId (TotalUpTimeChange upTime)
              repairTime <-
                randomExponentialProcess meanRepairTime
```

```
              machine

     runProcessInStartTime machine

     runEventInStopTime $
       return ()

-- | Основная ведущая модель.
masterModel :: Int -> Simulation DIO ()
masterModel n =
  do totalUpTime <- newRef 0.0

     let totalUpTimeChanged :: Signal DIO TotalUpTimeChange
         totalUpTimeChanged = messageReceived

     runEventInStartTime $
       handleSignal totalUpTimeChanged $ \x ->
       modifyRef totalUpTime (+ runTotalUpTimeChange x)

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (fromIntegral n * y)

     let rs =
           results
           [resultSource
            "upTimeProp"
            "The long-run proportion of up time (~ 0.66)"
            upTimeProp]

     printResultsInStopTime printResultSourceInEnglish rs

runSlaveModel :: (DP.ProcessId, DP.ProcessId)
                 -> DP.Process (DP.ProcessId, DP.Process ())
runSlaveModel (timeServerId, masterId) =
  runDIO m ps timeServerId
  where
    ps = defaultDIOParams { dioLoggingPriority = NOTICE }
    m  = do registerDIO
            runSimulation (slaveModel masterId) specs
            unregisterDIO

startSlaveModel :: (DP.ProcessId, DP.ProcessId) -> DP.Process ()
startSlaveModel x@(timeServerId, masterId) =
  do (slaveId, slaveProcess) <- runSlaveModel x
```

```
    slaveProcess

runMasterModel :: DP.ProcessId
                  -> Int
                  -> DP.Process (DP.ProcessId, DP.Process ())
runMasterModel timeServerId n =
  runDIO m ps timeServerId
  where
    ps = defaultDIOParams { dioLoggingPriority = NOTICE }
    m  = do registerDIO
            a <- runSimulation (masterModel n) specs
            terminateDIO
            return a

remotable ['startSlaveModel, 'curryTimeServer]

master = \backend nodes ->
  do liftIO . putStrLn $ "Slaves: " ++ show nodes
     let [n0, n1, n2] = nodes
         timeServerParams =
           defaultTimeServerParams { tsLoggingPriority = DEBUG }
     timeServerId <-
       DP.spawn n0
       ($(mkClosure 'curryTimeServer) (3 :: Int, timeServerParams))
     (masterId, masterProcess) <- runMasterModel timeServerId 2
     forM_ [n1, n2] $ \node ->
          DP.spawn node
          ($(mkClosure 'startSlaveModel) (timeServerId, masterId))
     masterProcess

main :: IO ()
main = do
  args <- getArgs
  case args of
    ["master", host, port] -> do
      backend <- initializeBackend host port rtable
      startMaster backend (master backend)
    ["slave", host, port] -> do
      backend <- initializeBackend host port rtable
      startSlave backend
  where
    rtable :: DP.RemoteTable
    rtable = __remoteTable initRemoteTable
```

Мы определили на языке Haskell отдельный тип сообщений и сделали его представителем класса типов `Serializable`. Для этого мы используем

возможность GHC автоматически выводить реализации классов типов.

Также мы вызываем функции на удаленных узлах. Cloud Haskell позволяет нам сделать это, для чего мы должны задействовать расширение Template Haskell.

При условии, что модель мы сохранили в файле *MachRep1.hs*, мы можем скомпилировать ее с помощью следующей команды в терминале[5].

```
$ ghc -O2 -threaded MachRep1.hs
```

Предполагается, что распределенная модель будет запущена на трех ведомых узлах и одном ведущем. В начале мы запускаем наши ведомые узлы. Для простоты мы запускаем узлы на одном локальном компьютере, который имеет IP-адрес 127.0.0.1, но мы могли бы задать и другие адреса IP из локальной сети.

```
$ ./MachRep1 slave 127.0.0.1 8080 &
$ ./MachRep1 slave 127.0.0.1 8081 &
$ ./MachRep1 slave 127.0.0.1 8082 &
```

Представленная модель действительно предполагает, что все узлы существуют в локальной сети и то, что ведущий узел может их найти без дополнительной конфигурации. В реальном же случае мы могли бы переписать модель и положить настоящие адреса IP в некоторый конфигурационный файл так, чтобы ведущий узел смог прочитать такой файл, а затем установить соединение с соответствующими удаленными узлами. Тогда не было бы ограничения на запуск всех узлов на локальном компьютере. Еще мы могли бы запустить через интернет.

Теперь для начала всей распределенной имитации мы запускаем ведущий узел, который инициирует всю работу.

```
$ ./MachRep1 master 127.0.0.1 8088
```

В своем случае я получил следующий вывод при условии, что здесь некоторая часть удалена была для краткости:

```
Slaves: [nid://127.0.0.1:8080:0,nid://127.0.0.1:8081:0,nid://127.0.0....
[INFO] Time Server: starting...
[DEBUG] Time Server: RegisterLogicalProcessMessage pid://127.0.0.1:80...
[DEBUG] Time Server: RegisterLogicalProcessMessage pid://127.0.0.1:80...
```

[5]Это старая команда, которая более не работает. Сейчас придется создавать полноценный проект Cabal или Stack. На май 2024 года сам код по-прежнему успешно компилируется.

```
[DEBUG] Time Server: RegisterLogicalProcessMessage pid://127.0.0.1:80...
[INFO] Time Server: starting
[DEBUG] Time Server: computing the global time...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8088:0:9 10000.0
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8082:0:10 0.735...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8081:0:10 0.231...
[INFO] Time Server: providing the global time = Just 0.2316644765668638
[DEBUG] Time Server: computing the global time...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8082:0:10 408.2...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8081:0:10 453.8...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8088:0:9 897.56...
[INFO] Time Server: providing the global time = Just 408.2912566906416
[DEBUG] Time Server: computing the global time...
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8082:0:10 10000.0
----------

-- simulation time
t = 10000.0

-- The long-run proportion of up time (~ 0.66)
upTimeProp = 0.6679609105729915

[DEBUG] Time Server: ComputeLocalTimeAcknowledgementMessage
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8088:0:9 10000.0
[DEBUG] Time Server: LocalTimeMessage pid://127.0.0.1:8081:0:10 10000.0
[INFO] Time Server: providing the global time = Just 10000.0
[DEBUG] Time Server: UnregisterLogicalProcessMessage pid://127.0.0.1:8...
[INFO] Time Server: providing the global time = Just 10000.0
[DEBUG] Time Server: UnregisterLogicalProcessMessage pid://127.0.0.1:8...
[INFO] Time Server: providing the global time = Just 10000.0
[ERROR] Exception occurred: ProcessTerminationException
[ERROR] Exception occurred: ProcessTerminationException
[DEBUG] Time Server: TerminateTimeServerMessage pid://127.0.0.1:8088:0:9
[INFO] Time Server: start terminating...
[INFO] Time Server: terminate
[ERROR] Exception occurred: ProcessTerminationException
```

В этом дампе мы видим, как глобальное время монотонно увеличивалось

до тех пор, пока оно не сравнялось с конечным модельным временем.

Если вы пользователь систем Linux или macOS, то вы можете запускать ведущий узел много раз, когда при каждом запуске распределенная имитация использует те же ведомые узлы. К сожалению, по некоторой неизвестной причине сетевые библиотеки работают нестабильно на Windows...[6]

Наконец, стоит отметить, что слово localhost, скорее всего, не распознается как допустимый адрес IP. Вместо него вам следует явно задавать 127.0.0.1.

12.5 Операции ввода/вывода

Операции ввода/вывода требуют особого подхода в оптимистичной распределенной имитации, потому что их нельзя откатить в общем случае. Если приходит запрос от модели на такую операцию, то логический процесс тут же пытается синхронизировать свое значение глобального виртуального времени. Процесс будет ожидать до тех пор, пока локальное время очереди событий не станет равным глобальному виртуальному времени. Сложный вопрос состоит в том, как обработать возможную ситуацию, когда произойдет откат во время такой синхронизации. К счастью, есть надежный и достаточно гибкий способ для обработки таких откатов, который мы и рассмотрим ниже в этом разделе.

Распределенные вычисления `Event` и `Process` являются представителями `MonadIO`.

```
instance MonadIO (Event DIO)
instance MonadIO (Process DIO)
instance MonadIO (Composite DIO)
```

Мы могли бы напрямую вызвать функцию `liftIO` из любого места распределенной имитации, но это было бы ненадежно по описанной выше причине. Если действительно возникает откат во время синхронизации глобального виртуального времени, и локальное время очереди по-прежнему остается больше глобального виртуального времени, то тогда возникла бы ошибка времени исполнения, что привело бы к прерыванию всей распределенной имитации. Решение состоит в том, чтобы вызывать `liftIO` только в надежных местах.

Между прочим, вызов `liftIO` в начальном модельном времени всегда надежен и быстр, поскольку нет необходимости в синхронизации. Невозможен

[6] Давно не проверялось. Сейчас положение дел могло и улучшиться на системе Windows.

откат, когда имитация только началась. Вызов действий `IO` в начальном модельном времени может быть полезен, если мы собираемся прочитать данные из файла или базы данных для инициализации имитации.

Существует очень похожий аналог функции `enqueueEvent`, который также регистрирует новое событие в заданное время.

```
enqueueEventIO :: EventIOQueueing m => Double -> Event m () -> Event m ()
```

Ключевая разница в том, что он гарантирует то, что локальное время очереди будет равно глобальному виртуальному времени при вызове соответствующего обработчика событий в заданное модельное время. Это в точности именно то, что нам нужно для безопасного запуска `liftIO`!

```
instance EventIOQueueing DIO
```

Как это работает? Функция `enqueueEventIO` также размещает новое событие в очереди. Однако перед активацией соответствующего обработчика события логический процесс начинает синхронизацию своего значения глобального виртуального времени. Процесс ждет до тех пор, пока локальное время очереди не сравняется с глобальным виртуальным временем. Если условие выполняется, то тогда вызывается исходный обработчик события, где мы можем безопасно вызывать `liftIO`, поскольку локальное время уже синхронизировано.

Однако, если все же происходит откат вместо успешной синхронизации, то тогда ничего особенного не применяется. Просто прерывается синхронизация и происходит обычный откат. Ничего такого не изменилось, никакое действие `IO` не было применено. Поэтому мы можем безопасно выполнить откат. После отката попытка вызвать соответствующий обработчик может быть снова повторена, если придет его время в следующий раз[7].

Итак, достаточно безопасно писать код вида

```
t <- liftDynamics time
liftEvent $
  enqueueEventIO t $
  liftIO $
  -- здесь мы могли бы прочитать из некоторого файла или записать в него
```

[7]Откат может быть и настолько глубоким, что будет откат и самого вызова `enqueueEventIO`. Тогда и в очередь не будет помещен соответствующий обработчик.

На самом деле, вычисление `Process` могло бы неявно создавать подобный код при каждом применении функции `liftIO`, но было решено так не делать, потому что порядок применения действий `IO` тогда был бы непредсказуем.

Наконец, из-за полной синхронизации действий `IO` строго рекомендуется использовать ссылки `Ref` вместо стандартных ссылок `IORef` везде, где только это возможно. Тип `Ref` много, много более легковесный и быстрый в распределенной имитации. Более того, ссылки `Ref` эффективно поддерживают откаты.

Чтобы у читателя сложилось правильное понимание, следует акцентировать внимание на том, что в языке Haskell побочные эффекты контролируются. Если не применять совсем уже запрещенных приемов с `unsafePerformIO`, то при соблюдении несложных правил можно ожидать с очень высокой степенью уверенности, что оптимистичная распределенная имитация с возможными откатами вернет правильный результат имитации. Это напрямую связано с тем, что побочные эффекты, включая операции ввода/вывода, контролируемы в языке Haskell. Это — одна из сильных сторон языка, которая оказалась очень кстати в случае распределенного имитационного моделирования.

12.6 Горизонт времени моделирования

Есть один неожиданный и иногда полезный эффект от использования функции `enqueueEventIO`. Вызывая эту функцию, мы можем ограничить *горизонт времени моделирования* в течение распределенной имитации, что задает, насколько далеко мы можем уйти во время имитации от некоторой точки модельного времени в прошлом, до которой еще возможен откат.

```
runEventInStartTime $
  enqueueEventIOWithTimes [100, 200..] $
  return ()
```

Здесь локальное время очереди будет синхронизировано с глобальным виртуальным временем в точках 100, 200, 300, … .

Однако существует гораздо более эффективный способ для определения горизонта моделирования. Мы можем явно задать поле `dioTimeHorizon` во время передачи `DIOParams` при запуске логического процесса. Существует общая рекомендация задавать горизонт моделирования всякий раз, когда это возможно, но его значение сильно зависит от заданной модели и того оборудования, где модель будет запущена.

Есть другой парадокс, когда ограничивая скорость имитации для каждого логического процесса, мы можем в целом улучшить общую производительность для всей распределенной имитации. Мы могли бы осознать это следующим образом. Введение явного горизонта моделирования ведет к меньшему числу откатов, а сами откаты становятся меньше по размеру, что может улучшить общую производительность имитации, а может, и нет. Правильный ответ не всегда должен быть интуитивно понятным.

Более того, тип `DIOParams` задает пороговые значения по размеру некоторых очередей. Помимо других очередей, определен журнал откатов действий, очередь отправленных сообщений и очередь для исходящих сообщений, которые логический процесс отослал, но еще не получил подтверждения о получении от другого логического процесса.

Заданы значения по умолчанию для этих пороговых величин. Мы можем их поменять до запуска новой распределенной модели. Так, если одно из пороговых значений превышено, то тогда логический процесс переходит в режим регулирования, когда он обрабатывает только пришедшие сообщения в надежде уменьшить размеры очередей. Это одновременно защита от перегрузки, но и потенциально слабое место, о котором вам следует знать.

Например допустим, что пороговый размер для очереди отправленных сообщений составляет 10000, но логический процесс зачем-то пытается отправить сразу 20000 сообщений одновременно в момент модельного времени $t = t_0$. Когда посылается 10000-е сообщение, логический процесс сразу же переходит в режим регулирования. К сожалению, он не может выйти из этого режима, заданного пороговым значением. Даже если глобальное время в итоге станет равным t_0, то тогда логический процесс по-прежнему не сможет удалить все отправленные 10000 сообщений из очереди отправленных сообщений. Если бы глобальное виртуальное время стало бы больше, чем t_0, то процесс смог бы, но сейчас глобальное виртуальное время не может превысить локальное время очереди. Таким образом, логический процесс навсегда остается в режиме регулирования. Решением было бы или увеличить пороговое значение для размера очереди, как минимум, сделав его равным 20001, или не посылать так много сообщений одновременно в то же самое модельное время t_0.

Итак, значение глобального виртуального времени очень велико в распределенной оптимистичной имитации. Невозможно откатиться до времени, которое было бы меньше глобального виртуального времени. Поэтому мы можем безопасно удалить все элементы очередей, которые бы соответствовали прошлому времени со значениями меньшими, чем глобальное виртуальное

время.

12.7 Повторный запуск вычислений

Распределенная имитация имеет другую ловушку, связанную с тем, что не все сообщения могут прийти вовремя или в строго заданном порядке. В итоге может сложиться такая необычная ситуация, когда логический процесс временно переходит в недопустимое состояние, а дальнейшее продолжение имитации теряет смысл до тех пор, пока мы не получим все сообщения, которые бы привели к откату, который бы в свою очередь выправил бы ситуацию и привел бы к правильному течению имитации.

Например, мы не можем освободить ресурс или многоканальное устройство GPSS, если они уже имеют максимально возможное содержимое. Если такая неправильная ситуация все же возникает, то тогда кидается особое исключение `SimulationRetry`, которое уже обрабатывается распределенным модулем Айвики. Симулятор временно переходит в особый режим, когда он только принимает сообщения в надежде, что они в итоге приведут к откату.

Мы можем искусственно вызвать такое исключение через вызов следующей функции, которая принимает в качестве параметра отладочное строковое сообщение для отображения в окне терминала, если все-таки попытка выправить вычисление окажется неудачной.

```
retryEvent :: MonadException m => String -> Event m a
```

Есть одно предупреждение касательно этого второго вида откатов. Если одно из сообщений имеет одинаковые время отправки и время получения, то тогда имитация рискует впасть в бесконечный цикл. Возможно, что это просто ограничение текущей реализации.

По крайней мере, если время получения больше времени отправки сообщения, будь их разница хотя бы одной миллионной, то тогда откаты этого второго типа успешно обрабатываются. Поэтому строго рекомендуется использовать функцию `enqueueMessage` вместо `sendMessage`.

12.8 Восстановление после ошибок соединения

Айвика способна восстановить распределенную имитацию после временных ошибок сетевого соединения, но этот режим должен быть активирован явно в конфигурационных параметрах.

Например, если мы возьмем модель из раздела 12.4, то тогда мы должны будем внести следующие изменения:

```
runSlaveModel :: (DP.ProcessId, DP.ProcessId)
                 -> DP.Process (DP.ProcessId, DP.Process ())
runSlaveModel (timeServerId, masterId) =
  runDIO m ps timeServerId
  where
    ps = defaultDIOParams { dioLoggingPriority = NOTICE,
                            dioProcessMonitoringEnabled = True,
                            dioProcessReconnectingEnabled = True }
    m  = do registerDIO
            runSimulation (slaveModel masterId) specs
            unregisterDIO

startSlaveModel :: (DP.ProcessId, DP.ProcessId) -> DP.Process ()
startSlaveModel x@(timeServerId, masterId) =
  do (slaveId, slaveProcess) <- runSlaveModel x
     DP.send slaveId (MonitorProcessMessage timeServerId)
     DP.send masterId (MonitorProcessMessage slaveId)
     DP.send slaveId (MonitorProcessMessage masterId)
     slaveProcess

runMasterModel :: DP.ProcessId
                  -> Int
                  -> DP.Process (DP.ProcessId, DP.Process ())
runMasterModel timeServerId n =
  runDIO m ps timeServerId
  where
    ps = defaultDIOParams { dioLoggingPriority = NOTICE,
                            dioProcessMonitoringEnabled = True,
                            dioProcessReconnectingEnabled = True }
    m  = do registerDIO
            a <- runSimulation (masterModel n) specs
            terminateDIO
            return a

remotable ['startSlaveModel, 'curryTimeServer]

master = \backend nodes ->
  do liftIO . putStrLn $ "Slaves: " ++ show nodes
     let [n0, n1, n2] = nodes
         timeServerParams =
           defaultTimeServerParams {
             tsLoggingPriority = DEBUG,
             tsProcessMonitoringEnabled = True,
```

```
        tsProcessReconnectingEnabled = True }
timeServerId <-
  DP.spawn n0
  ($(mkClosure 'curryTimeServer) (3 :: Int, timeServerParams))
(masterId, masterProcess) <- runMasterModel timeServerId 2
DP.send masterId (MonitorProcessMessage timeServerId)
forM_ [n1, n2] $ \node ->
     DP.spawn node
     ($(mkClosure 'startSlaveModel) (timeServerId, masterId))
masterProcess
```

Здесь мы активируем для каждого логического процесса булевы параметры `dioProcessMonitoringEnabled` и `dioProcessReconnectingEnabled`. Похожим образом мы активируем параметр `tsProcessMonitoringEnabled` и аналогичный параметр `tsProcessReconnectingEnabled` для сервера времени. Также нам нужно послать сообщение `MonitorProcessMessage` для начала отправки сообщений о сохранении активности каждого из сетевых соединений, указав соответствующие идентификаторы отправителя и получателя. Для сервера времени достаточно односторонней проверки.

В таком случае распределенная имитация способна восстановить себя после временных ошибок сетевого соединения. Здесь мы полагаем, что сетевые соединения более подвержены сбою, чем сами вычислительные узлы кластера.

Однако имитация не может ждать бесконечно долго логических процессов. А что если один из узлов кластера вовсе выключился? Мы можем найти ответ в следующем разделе.

12.9 Остановка разъединенной имитации

При ошибках разъединения сети будут задействованы интервалы по времени, в течение которого распределенная имитация будет ожидать другие логические процессы перед тем, как посчитать соединение с ними потерянным. Это необходимо для автоматической остановки распределенной имитации, если кластер ломается в течение достаточно долгого времени по некоторой причине. Один из узлов кластера мог вовсе быть перезапущен, или он мог быть выключен. Тогда сетевое соединение может быть утеряно навсегда.

Подход достаточно прост. Поскольку сервер времени является единой точкой сбоя, то достаточно проверять соединение с сервером времени, которое уникально для каждого запуска распределенной имитации.

По умолчанию, если сервер времени не получал каких либо сообщений от одного из логических процессов в течение интервала времени, заданного конструктором данных `TerminateDueToLogicalProcessTimeout` (по умолчанию 5 минут), то тогда сервер времени останавливает самого себя. К этому интервалу мы обычно должны добавить временной интервал для синхронизации сессии, который задается параметром `tsTimeSyncTimeout` (по умолчанию 1 минута). Итого, после 6 минут (= 5 + 1) сервер времени должен остановить самого себя.

Для логического процесса существуют аналогичные прежним конструктор данных `TerminateDueToTimeServerTimeout` и параметр `dioSyncTimeout` с теми же самыми значениями по умолчанию. Это означает, что если логический процесс не получал никаких сообщений от сервера времени в течение заданных временных интервалов, то тогда логический процесс остановит самого себя. По умолчанию он тоже остановится через 6 минут (= 5 + 1).

Другими словами, если кластер сломался, то его последная живая часть остановит саму себя, как минимум, через 12 минут (= 6 + 6). Эти значения временных интервалов можно менять и настраивать.

Поэтому очень важно передавать сообщения о сохранении активности соединения, режим которых должен быть явно активирован посылкой сообщений `MonitorProcessMessage` в самом начале каждого запуска распределенной имитации. Тогда вам следует также и активировать параметры мониторинга и восстановления, как было описано в разделе 12.8.

12.10 Распределенная имитация как сервис

В продолжение предыдущих двух разделов мы поговорим о создании сервисов для моделирования.

Итак, распределенная имитация может восстанавливать себя после временных сетевых ошибок соединения. Однако, если разъединение длилось слишком долго, то тогда все части распределенного имитационного кластера автоматически остановят сами себя после истечения некоторых временных интервалов, которые задают, как долго мы можем ожидать логические процессы в случае возникновения ошибок сетевого соединения.

Даже если ведомый логический процесс остановится, то тогда его вычислительный узел останется активным, и он сможет принять новый запуск имитации. Однако, если компьютер узла был перезагружен, то тогда сам узел может быть заново запущен, например как сервис операционной системы

Linux, и узел снова станет доступным для запуска новой имитации. Если же что-то серьезное произошло, когда узел не может восстановиться, то тогда обычно это решается на другом уровне через дублирование соответствующих компьютерных узлов.

Что касается ведущего логического процесса, то он работает по принципу *все-или-ничего*. Либо ведущий логический процесс вернет результат, и это будет корректный результат, потому что имитация аналитическая, либо ведущий логический процесс завершится некоторой ошибкой. В последнем случае мы можем начать новый имитационный запуск после некоторой задержки в надежде, что ведомые узлы восстановятся к тому времени.

Все это позволяет на основе распределенного кластера создавать сервисы для моделирования с заданными характеристиками на основе операционной системы Linux.

12.11 Мониторинг распределенной имитации

Мы могли бы вести мониторинг за распределенной имитацией, наблюдая за сообщениями из логов, но разумнее и предпочтительнее в лог писать только ошибки и предупреждения, тогда как сам мониторинг может быть реализован на другом уровне.

Существуют дополнительные функции запуска для сервера времени и логического процесса. Они позволяют нам задать обработчики, которые бы уже что-то делали с информацией по мониторингу, вероятно, отсылая ее на некоторый внешний специализированный сервис.

```
timeServerWithEnv :: Int
                     -> TimeServerParams
                     -> TimeServerEnv
                     -> DP.Process ()
```

Эта функция запускает новый сервер времени, где тип `TimeServerEnv` позволяет задать функцию, которая будет вызвана из отдельного процесса каждый раз, как симулятор Айвики пришлет данные по мониторингу.

```
data TimeServerEnv =
  TimeServerEnv { tsSimulationMonitoringAction ::
                       Maybe (TimeServerState -> DP.Process ())
                }
```

Тип данных `TimeServerState` описывает текущее состояние сервера времени. Оно включает в себя глобальное виртуальное время и список зарегистрированных логических процессов. В обработчике вы можете отослать эту информации далее на специализированный сервис мониторинга, или просто напечатать в консоли через стандартный вывод, или записать в некоторый файл.

Интервал физического времени между сеансами получения данных мониторинга задается параметром `tsSimulationMonitoringInterval` типа данных `TimeServerParams`, значение которого вы можете задать перед запуском сервера времени.

Похожим образом мы можем запустить логический процесс, задав параметр типа данных `DIOEnv`.

```
runDIOWithEnv :: DIO a -> DIOParams -> DIOEnv -> DP.ProcessId
                 -> DP.Process (DP.ProcessId, DP.Process a)
```

Тип `DIOEnv` также содержит обработчик, который уже предназначен для мониторинга состояния логического процесса.

```
data DIOEnv =
  DIOEnv { dioSimulationMonitoringAction ::
              Maybe (LogicalProcessState -> DP.Process ())
         }
```

Тип данных `LogicalProcessState` описывает текущее состояние логического процесса. Оно включает в себя локальное время процесса, локальное время очереди, размер очереди событий, размер журнала откатов действий, количество входящих сообщений, количество отправленных сообщений, количество исходящих сообщений, доставка которых еще не была подтверждена, а также общее количество выполненных откатов. В обработчике вы можете переслать эту информацию далее на специализированный сервис, или напечатать в консоли через стандартный вывод, или также записать в некоторый файл.

Интервал физического времени между сеансами получения данных мониторинга задается через параметр `dioSimulationMonitoringInterval` типа данных `DIOParams`, значение которого вы можете определить до запуска логического процесса.

Таким образом, симулятор предоставляет нам средства активного мониторинга за текущим состоянием распределенной имитации.

12.12 Распределенный вычислительный эксперимент

Как и прежде, под названием вычислительного эксперимента мы подразумеваем имитационный эксперимент по методу Монте-Карло, где мы создаем веб-страницу с результатами моделирования, представленными в виде графиков, гистограмм, сводной статистики, ссылок на таблицы в формате CSV и тому подобное.

Распределенный вычислительный эксперимент сложен уже самим тем, что применяется оптимистичный метод, что подразумевает собой множество откатов в течение имитации. Более того, имитация распределена, а узлы кластера могут располагаться на физически разных компьютерах.

Поэтому предлагается следующих подход, который будет работать для любого вида моделирования: и последовательной, и распределенной. Итоговые данные обрабатываются в конце каждого запуска имитации[8]. Данные сохраняются в таблицы SQL. После окончания имитационного эксперимента мы строим отчет с результатами моделирования на основе данных, сохраненных в таблицах SQL.

Для этого нам нужны дополнительные библиотеки, включенные в состав *Aivika Extension Pack*. Смотрите приложение A для инструкций по установке. Однако это выходит за рамки этой книги. Поэтому распределенный вычислительный эксперимент здесь не рассматривается. Мы только заметим, что есть такая возможность.

12.13 Заключение

Айвика позволяет нам запускать параллельные и распределенные дискретно-событийные имитационные модели на основе метода деформации времени. Имитация может быть запущена как на одном компьютере, так и в облачном сервисе или на распределенном кластере. Разные сетевые протоколы поддерживаются в Cloud Haskell, который Айвика использует для соединения логических процессов. Например компьютеры могут быть соединены через обычный интернет.

[8] Есть возможность обрабатывать частями по мере продвижения модельного времени, если периодически выполнять синхронизацию глобального виртуального времени. См. раздел 12.5.

Распределенная имитация способна восстанавливать себя после временных ошибок сетевого соединения, но заложен встроенный механизм остановки имитации после истечения интервалов по времени ожидания. Так мы можем строить сервисы для моделирования с желаемыми характеристиками. Более того, мы можем вести мониторинг за такой распределенной имитацией в реальном времени.

Помимо тех замечательных преимуществ, что дает нам распределенная имитация, стоит еще раз заметить, что этот тип моделирования отнюдь не бесплатен. В большинстве случаев последовательная модель будет много быстрее и много проще для использования. Передача сообщений и запуск распределенной модели — это дорогие операции. Запуск связан с медленной инициализацией сетевых сервисов, тогда как передача сообщений дорога сама по себе, но еще и потенциально является источником откатов в имитации. Поэтому должна быть серьезная причина, почему стоит выбрать распределенную модель последовательной. Либо модель слишком огромна, чтобы вместиться в памяти одного компьютера, либо мы в состоянии получить реальный выигрыш от параллелизма, запуская множество параллельных или распределенных логических процессов.

Тем не менее, это так увлекательно, что мы можем строить и последовательные, и распределенные имитационные модели на основе единого унифицированного подхода, который предоставляет Айвика! Как мы вскоре увидим, тот же самый подход работает и в случае вложенного моделирования. Например, везде мы можем использовать события, агенты, дискретные процессы, вообще, использовать все построенное поверх них, включая GPSS-подобный DSL, описанный в главе 10.

Часть III

Вложенное моделирование

Эта часть книги посвящена вложенному моделированию, когда мы можем запускать имитации внутри имитаций, затем новые имитации внутри тех имитаций внутри имитаций и так далее рекурсивно. Эта часть может быть разделена на два типа вложенного моделирования.

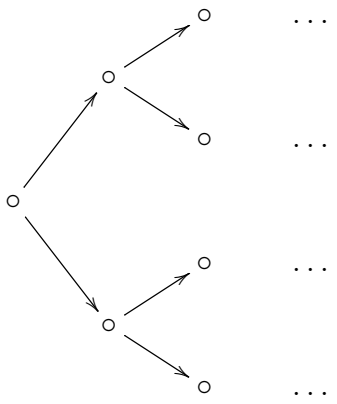
Первый тип предполагает более гибкий подход, когда мы можем делать ответвления вложенной модели в любой момент модельного времени, но это может привести, чаще всего, к экспоненциальному росту числа имитаций внутри начальной исходной имитации. Поэтому обычно нам следует ограничивать рост дерева имитаций некоторой глубиной.

Второй подход использует особую изощренную структуру вместо дерева. Эта структура известна как решетка. Здесь вложенные имитации создаются только в ограниченном числе узлов решетки, но зато мы можем обойти такие узлы за квадратичное время. Например это может быть полезно для финансовой математики.

Глава 13

Ветвление

Сначала мы рассмотрим тип вложенного моделирования, когда мы можем делать ответвления вложенной имитации в любой момент модельного времени. Это соответствует дереву имитаций, которое потенциально имеет экспоненциальный рост числа узлов чаще всего.



13.1 Вычисление ветвящихся имитаций

Пакет `aivika-branches` реализует соответствующий тип вложенного моделирования. Он определяет моделирующее вычисление `BR`, которое само следует параметризовать монадой `IO` для того, чтобы вычисление стало представителем класса типов `MonadDES` из главы 11.

```
module Simulation.Aivika.Branch
```

```
data BR m a

instance MonadDES (BR IO)
```

Вычисление имеет очень простую функцию запуска, которая возвращает значение в соответствующей монаде, по которой вычисление было параметризовано.

```
runBR :: MonadIO m => BR m a -> m a
```

Мы скоро увидим, как создавать дерево вложенных имитаций. Нам понадобится ограничить это дерево некоторой глубиной. Поэтому нам нужна функция, который бы возвращала текущий уровень вложенности, начиная с нуля.

```
branchLevel :: Monad m => BR m Int
```

Корневая исходная имитация имеет уровень 0. Следующие имитации, созданные непосредственно из корня, будут иметь уровень 1. Затем последующие вложенные имитации, созданные непосредственно из тех первых, будут иметь уровень 2 и так далее.

Наступило время показать то, как мы можем создавать вложенные имитации.

```
branchEvent :: Event (BR IO) a -> Event (BR IO) a
```

Функция `branchEvent` ответвляет текущую вложенную имитацию, запускает заданное вычисление в рамках ответвления в то же самое модельное время, а затем возвращает результат родительской имитации.

Крайне важно то, что вложенная имитация не может никоим образом изменить ни очередь событий родительской имитации, ни одно из значений ссылок `Ref` родителя, то есть вложенная имитация не может изменить состояние модели из родительской имитации.

Это еще одна причина, почему вам следует использовать тип ссылок `Ref` вместо стандартной ссылки `IORef` всякий раз, когда это возможно. Хотя тип `IORef` все же может оставаться полезным, например для включения некоторого триггера при обнаружении чего-то существенного во вложенной имитации, но число таких сценариев довольно-таки ограниченно.

Можно думать о новом ответвлении как о клонировании состояния модели из текущей имитации. Создание новой ветви — очень дешевая и быстрая

операция, но не бесплатная, тем не менее. Клон наследует состояние своего родителя. События, которые должны были быть активированы, будут также активированы, но уже в рамках вложенной имитации без какого-либо воздействия на родительскую имитацию. Ссылка `Ref` наследует свое состояние от родительской имитации, но любое изменение ссылки `Ref` не изменит никоим образом ни значения родительской имитации, ни ее предков, если таковые были. У корневой имитации предков нет.

Более того, мы можем делать новые ответвления вложенных имитаций уже внутри ответвленной имитации. Каждый раз, как мы создаем новую вложенную имитацию, мы увеличиваем на единицу уровень вложенности относительно родительской имитации.

Может быть полезно запустить ответвленную имитацию с некоторой задержкой по модельному времени, так чтобы все ожидающие события были обработаны в рамках вложенной имитационной модели. Поэтому определена другая функция, которая позволяет задать точное время, в которое нам следует запустить заданное вычисление в будущем модели, хотя концептуально эта функция очень похожа на `branchEvent`, только модельное время запуска отличается:

```
futureEvent :: Double -> Event (BR IO) a -> Event (BR IO) a
```

Здесь первый параметр задает время активации вычисления, но результат возвращается в родительскую имитацию в текущее модельное время, как и было прежде. Важно то, что родительская имитация затем продолжает свою работу в текущем модельном времени. Поэтому, кстати, здесь мы имеем вычисление `Event`, а не `Process`.

13.2 Пример: ветвление имитаций

Для демонстрации использования вложенного моделирования мы рассмотрим довольно-таки концептуальный пример, где мы оценим случайную величину, усредняя ее оценки, сделанные в будущем времени. Как обычно, мы будем использовать уже привычную нам модель из раздела 11.3.

> Есть два станка, которые иногда ломаются. Время наработки распределено экспоненциально со средним 1.0, и время восстановления распределено экспоненциально со средним 0.5. Есть двое

мастеров. Так что, оба станка могут быть отремонтированы одновременно, если они сломались в то же время. Вывод задает долговременную пропорцию времени наработки. Должны получить значение около 0.66.

Модель на языке Haskell приведена ниже. Она достаточно проста.

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

import Simulation.Aivika.Trans
import Simulation.Aivika.Branch

meanUpTime = 1.0
meanRepairTime = 0.5

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

maxLevel = 10

delta :: Int -> Parameter (BR IO) Double
delta n =
  do t0 <- liftParameter starttime
     t2 <- liftParameter stoptime
     return $ (t2 - t0) / fromIntegral n

forecast :: Event (BR IO) Double -> Event (BR IO) Double
forecast m =
  do let loop dt0 i =
           do level <- liftComp branchLevel
              if level <= maxLevel
                then do t  <- liftDynamics time
                        x1 <- futureEvent (t + dt0) $ loop dt0 (i - 1)
                        x2 <- futureEvent (t + dt0) $ loop dt0 (i + 1)
                        let x = (x1 + x2) / 2
                        x `seq` return x
                else m
     dt0 <- liftParameter $ delta maxLevel
     loop dt0 0

model :: Simulation (BR IO) (Results (BR IO))
```

```
model =
  do totalUpTime <- newRef 0.0

     let machine =
           do upTime <-
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)
              repairTime <-
                randomExponentialProcess meanRepairTime
              machine

     runProcessInStartTime machine
     runProcessInStartTime machine

     let upTimeProp =
           do x <- readRef totalUpTime
              t <- liftDynamics time
              return $ x / (2 * t)

     upTimePropForecasted <-
       runEventInStartTime $
       forecast upTimeProp

     return $
       results
       [resultSource
        "upTimeProp"
        "The long-run proportion of up time (~ 0.66)"
        upTimeProp,
        --
        resultSource
        "upTimePropForecasted"
        "The forecasted long-run proportion of up time"
        (return upTimePropForecasted :: Event (BR IO) Double)]

main :: IO ()
main =
  runBR $
  printSimulationResultsInStopTime
  printResultSourceInEnglish
  model specs
```

Обратите внимание на то, как мы оцениваем в вспомогательной функции `forecast` заданное вычисление `Event`, разделяя его на две части, а затем получая среднее значение. Мы делаем это рекурсивно так, чтобы глубина

дерева не превысила значения 10, начиная отсчет от 0. Здесь мы создаем $2^{10+1} - 1 = 2047$ имитаций, включая саму исходную корневую имитацию.

Также значение переменной `upTimePropForecasted` является чистым значением типа `Double`. Поэтому мы оборачиваем его в вычисление `Event` для того, чтобы мы могли вернуть его как источник результата.

При запуске модели я получил следующий вывод:

```
$ time ./MachRep1
----------

-- simulation time
t = 1000.0

-- The long-run proportion of up time (~ 0.66)
upTimeProp = 0.6784040642495368

-- The forecasted long-run proportion of up time
upTimePropForecasted = 0.6636117393274816


real 0m3.845s
user 0m3.778s
sys 0m0.039s
```

Как мы можем увидеть, 2047 имитаций длились около 4-х секунд. Корневая имитация прошла весь интервал по модельному времени от начала и до конца, тогда как вложенные имитации были краткоживущими с временным интервалом равным $(1000 - 0)/10 = 100$ единицам времени. Полученная через заглядывание в будущее модели оценка кажется более точной, чем полученное первое значение, которое соответствует тому, как если бы имитация была только последовательной.

13.3 Заключение

Создавая новые ответвления, мы можем запускать вложенные модели в любой момент модельного времени и делать это настолько часто, насколько мы хотели бы. Это позволяет нам предсказывать будущее поведение текущей имитации. Более того, это могло бы быть дополнением к методу Монте-Карло, поскольку сейчас мы можем увеличить общее число имитаций, используемых для оценки случайных величин.

Однако наиболее важной проблемой ветвящихся имитаций является то, что они могут иметь экспоненциальную сложность и даже хуже относительно глубины дерева вложенных моделей. Чем больше мы заглядываем в будущее модели, тем больше узлов дерева мы должны обойти, что в итоге может привести к комбинаторному взрыву.

В следующей главе мы рассмотрим другой тип вложенного моделирования, где сложность обхода узлов уже квадратичная, что позволит нам заглянуть далеко в будущее модели. Только нам следует применять этот тип вложенного моделирования с большой осторожностью, четко представляя себе, где он применим.

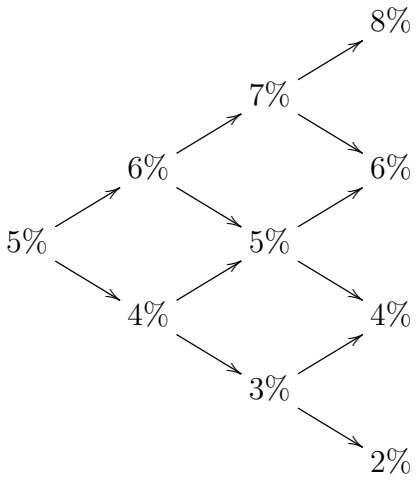
Глава 14

Решетка

Описанное ниже в этой главе является довольно-таки экспериментальным. Была красивая идея, которая широко распространена в финансовом моделировании. Я был так впечатлен этой идеей, что решил применить ее для дискретно-событийного моделирования. Пожалуйста, отнеситесь к описанному ниже методу под призмой критического анализа.

14.1 Введение решетки

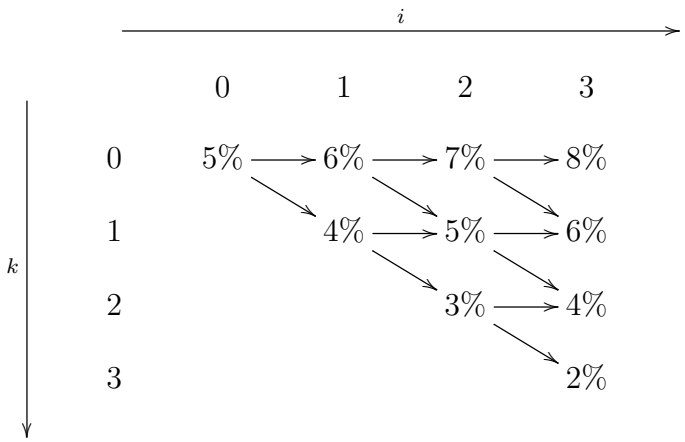
Начнем с финансового моделирования[11], рассмотрев изменение краткосрочной процентной ставки.



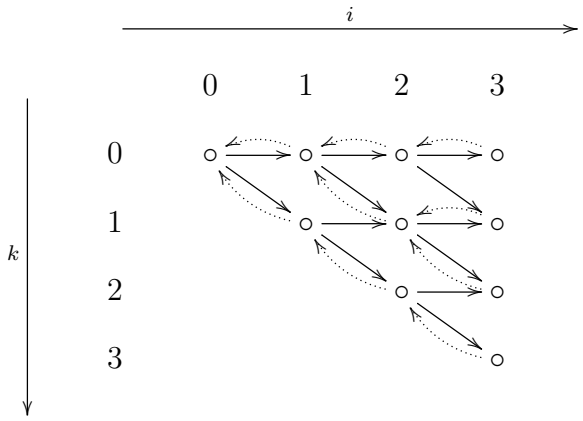
Обратите внимание на то, что некоторые узлы имеют обоих родителей.

Подобная структура называется *решеткой* (*англ.* lattice). Она имеет линейный рост ширины структуры по мере увеличения ее глубины, что позволяет нам обойти все узлы за вычислительно достижимое реальное время.

Теперь для дальнейшего изложения мы слегка изменим решетку. Это будет та же самая структура, что и прежде. Только узлы расположены иначе. Также изменена система координат.



Теперь вообразим, что узлы решетки представляют собой вложенные имитационные модели или же представляют саму исходную имитацию в случае корня. Поскольку каждая вложенная имитация должна быть порождена по некоторой единственной родительской имитации, то мы случайным образом выберем того единственного предка для внутренних узлов, от которого этот узел непосредственно порожден — для обозначения такого отношения мы будем использовать точечные стрелки. Как и прежде, внутренние узлы могут иметь двух родителей, но в действительности узлы могут быть порождены только от одного из них.



Для примера мы здесь сделали некоторый произвольный выбор, такой что вложенная имитация с координатами $(i, k) = (2, 1)$ имеет двух порожденных дочерних имитаций в координатах $(3, 1)$ и $(3, 2)$. В то же самое время возможны вложенные имитации, которые бы не имели дочерних порожденных. Многие из представленных на рисунке узлов имеют одну порожденную дочернюю имитацию или сразу две.

Строго говоря, мы можем сказать, что дочерняя вложенная имитация *порождена* соответствующей родительской имитацией, если первая является ответвлением последней в том смысле, как мы понимали это в главе 13.

Очевидно, что при выборе того, какие узлы от каких порождены, мы теряем информацию. Однако делая такое предположение, что каждый внутренний узел может иметь до двух родительских узлов, мы применяем трюк, который и позволяет нам обойти все узлы за квадратичное время относительно глубины решетки. Как вы помните, все узлы представляют собой вложенные имитационные модели за исключением корневого узла, который уже представляет собой саму исходную имитацию.

Назовем координату i как *индекс времени* (*англ.* time index), тогда как координата k пусть называется *индекс элемента* (*англ.* member index). При том условии, что размер (глубина) решетки равен n, мы получаем следующие ограничения:

$$\begin{aligned} &0 \le i \le n, \qquad (n > 0) \\ &0 \le k \le i. \end{aligned}$$

Индекс времени $i = 0$ будет соответствовать началу имитации, а индекс $i = n$ будет соответствовать конечной точке моделирования. Другие значения

индекса времени будут соответствовать точкам модельного времени в узлах сетки с заданным шагом.

$$t_i = \texttt{starttime} + i * \frac{(\texttt{stoptime} - \texttt{startime})}{n}$$

Если вложенная имитация имеет координаты (i, k), то тогда ее завершающее модельное время будет в точности равно t_i, в которое такая имитация пришла из времени t_{i-1}.

Корневая имитация является исключением. Она может пройти по всему временному интервалу от начала и до конца, как это было в случае другого типа вложенного моделирования, рассмотренного в предыдущей главе. Все вложенные имитации в конечном итоге порождаются от корневой имитации, когда она имела модельное время равное начальному времени. Поэтому перед созданием ответвлений корневая имитация уже должна содержать все необходимые будущие события.

Общее количество *всех* узлов решетки равно

$$\frac{(1 + (n + 1))}{2} \times (n + 1) \leq (n + 1)^2,$$

что делает обход узлов решетки вычислительно достижимой задачей даже для достаточно больших значений n[1].

Например, если $n = 1000$, то мы будем иметь только 501501 имитационную модель, что не так много для современных компьютеров, особенно, если 501500 вложенных имитаций из их числа являются краткоживущими!

14.2 Тип данных решетки

Экспериментальная реализация представлена пакетом `aivika-lattice`, который экспортирует определения в следующем модуле.

```
module Simulation.Aivika.Lattice
```

Решетка представлена типом данных `LIOLattice`. Для ее создания мы должны передать размер, а также функцию, которая бы возвращала индекс

[1] Здесь мы больше упираемся в то, а насколько нам хватит ОЗУ, чтобы держать в памяти столько вложенных имитаций.

элемента узла родительской имитации по заданным индексу времени и индексу элемента узла дочерней имитации. Тогда дочерняя вложенная имитация будет порождена по заданной родительской имитации через создание ответвления последней.

```
lattice :: Int
           -- ^ размер (глубина) решетки
           -> (Int -> Int -> Int)
           -- ^ получить индекс элемента узла родителя
           -- по заданным индексу времени и индексу элемента
           -> LIOLattice
```

К счастью, есть более удобная функция, котора позволяет порождать случайные решетки с заданной вероятностью и заданным размером, где значение вероятности определяет то, будет ли внутренний дочерний узел порожден от правого родителя. Вторая функция просто вызывает первую с вероятностью 0.5.

```
newRandomLatticeWithProb :: Double -> Int -> IO LIOLattice

newRandomLattice :: Int -> IO LIOLattice
newRandomLattice = newRandomLatticeWithProb 0.5
```

На самом деле, решетка связана с биномиальной случайной величиной. А какую вероятность использовать — это предмет анализа, который зависит от заданной модели.

14.3 Моделирующее вычисление решетки

Пакет `aivika-lattice` определяет монаду `LIO`, которая может быть использована для запуска вложенных имитаций в узлах решетки.

```
data LIO a

instance Monad LIO
instance MonadIO LIO
instance MonadDES LIO
```

Поскольку `LIO` является представителем `MonadDES`, то мы можем параметризовать обобщенную версию Айвики этим вычислением. Как уже упоминалось ранее, этот факт буквально означает то, что мы можем определять

события, дискретные процессы, агенты и GPSS-подобные блоки в рамках вложенных имитаций, которые бы запускались в узлах решетки.

Мы можем запустить вычисление `LIO` по заданному объекту решетки.

```
runLIO :: LIOLattice -> LIO a -> IO a
```

Каждому узлу решетки соответствует некоторый индекс времени и индекс элемента, как было определено в разделе 14.1. Мы можем запросить значения этих индексов в рамках вычисления `LIO`, которое выполняется в текущем узле решетки.

```
latticeTimeIndex :: LIO Int
latticeMemberIndex :: LIO Int
latticeParentMemberIndex :: LIO (Maybe Int)
```

Функция `latticeParentMemberIndex` возвращает индекс элемента родительского узла решетки, от которого текущий узел порожден, то есть ответвлен. Исключением является корневой узел, у которого нет родителя, поскольку он представляет собой саму исходную имитацию. Родительский узел всегда имеет индекс времени равный текущему индексу времени минус один.

Мы также можем запросить размер решетки, то есть глубину структуры решетки.

```
latticeSize :: LIO Int
```

Чтобы получить завершающее модельное время, привязанное к текущему узлу решетки, мы можем вызвать следующую функцию[2].

```
latticeTime :: Parameter LIO Double
```

Может быть полезно получить список всех значений времени, привязанных к узлам решетки, например для того, чтобы реализовать биномиальную модель ценообразования опционов.

```
latticeTimes :: Parameter LIO [Double]
```

Есть удобная для использования функция, которая активирует заданный обработчик во всех узлах решетки с соответствующим завершающим временем.

```
enqueueEventWithLatticeTimes :: Event LIO () -> Event LIO ()
```

[2]Вычисление `time` может вернуть и меньшее время, если вложенная имитация еще не достигла своего завершения.

14.4 Вычисление для наблюдений

После того, как имитация завершится, мы получим множество данных, которые еще должны быть обработаны для оценки характеристик случайных величин. Вложенное моделирование внутри решетки — это, главным образом, оценка таких характеристик.

Существует промежуточное вычисление, которое является мостом между моделирующими о оценивающими блоками. Это класс типов `Observable` из пакета `aivika-transformers`.

```
class Observable o m where
  readObservable :: o a -> m a
```

Некоторый тип данных является представителем класса `Observable`, если мы можем прочитать данные внутри некоторого вычисления. Обратите внимание на то, что мы параметризуем класс типов некоторым вычислением, в рамках которого мы можем прочитать текущее значение наблюдаемой величины.

Важно, что ссылка `Ref` является представителем `Observable` в рамках вычисления, которое мы введем в следующем разделе. Тогда наша дискретно-событийная модель будет сама изменяться и обновлять ссылки `Ref`, которые мы позже сможем прочитать во время последующей оценки некоторых случайных значений.

14.5 Вычисление для оценки

Блок оценки основан на использовании вычисления монадического трансформера `Estimate`.

```
data Estimate m a

instance MonadTrans Estimate
instance Monad m => Monad (Estimate m)
instance MonadIO m => MonadIO (Estimate m)
```

Монада `Estimate` является в точности тем вычислением, в рамках которого мы можем читать текущие значения ссылки `Ref`.

```
instance Observable (Ref LIO) (Estimate LIO)
```

Имеет смысл запускать вычисление `Estimate` только в начальной точке моделирования. От этой точки мы можем рекурсивно обойти все узлы.

```
runEstimateInStartTime :: MonadDES m => Estimate m a -> Simulation m a
```

Мы всегда можем запросить текущее модельное время в рамках вычисления `Estimate`. Вычисление `Esimate` запускается в точности в узлах решетки с использованием завершающего времени таких узлов.

```
estimateTime :: MonadDES m => Estimate m Double
```

Существуют разные комбинаторы, и наиболее важным из них является следующий. Он вызывает переданное вычисление `Estimate` в узлах решетки по заданным индексу времени и индексу элемента. Таким способом мы можем запросить значение как для будущих, так и прошлых узлов.

```
estimateAt :: Int
              -- ^ индекс времени для решетки
              -> Int
              -- ^ индекс элемента для решетки
              -> Estimate LIO a
              -- ^ вычисление
              -> Estimate LIO a
```

Есть удобная для использования функция `foldEstimate`, которая позволяет находить свертку некоторого вычисления по заданной функции преобразования, которая бы применялась к промежуточным узлам решетки. Полученные после оценки значения в двух дочерних узлах передаются этой преобразующей функции до тех пор, пока мы не дойдем до исходного корневого узла. До этого само вычисление вызывается в листовых узлах решетки с конечным временем моделирования, откуда и начинается процесс сворачивания вычисления.

```
-- | Свертка для оценки некоторого вычисления.
foldEstimate :: (a -> a -> Estimate LIO a)
                -- ^ свернуть два промежуточных узла решетки
                -> Estimate LIO a
                -- ^ оценить вычисление в конечной точке моделирования
                -> Simulation LIO (Estimate LIO a)
```

Вызвав функцию `foldEstimate` для корневого узла решетки, мы в итоге обойдем все узлы решетки.

Например, это позволяет нам реализовать оператор *snell*[11], который может быть полезен для оценивания цены опционов и контрактов в финансовом моделировании.

Сама функция `foldEstimate` использует комбинатор, который позволяет мемоизировать уже вычисленные значения в узлах решетки.

```
memoEstimate :: (Estimate LIO a -> Estimate LIO a)
                -- ^ оценить в промежуточных узлах решетки
                -> Estimate LIO a
                -- ^ оценить в листовых узлах с конечными точками времени
                -> Simulation LIO (Estimate LIO a)
```

Этот комбинатор позволяет нам создавать рекурсивные вычисления в узлах решетки.

14.6 Пример: биномиальное распределение

Как было замечено ранее, решетка сильно связана с биномиальной случайной величиной. Давайте попробуем оценить влияние листовых узлов решетки в предположении о том, что оба дочерних узла равновероятны. Легко увидеть, что индекс элемента будет иметь биномиальное распределение с параметром $p = 0.5$.

Сейчас мы запишем имитационную модель, чтобы убедится в этом через численный эксперимент. Для построения гистограммы мы будем использовать дополнительные функции из пакета `aivika-experiment`. Только, пожалуйста, заметьте, что использованный ниже метод построения гистограммы имеет экспоненциальную сложность в отличие от самого процесса обхода узлов, который уже имеет квадратичную сложность.

Поэтому не рекомендуется повторять этот эксперимент с большими размерами решетки. Вместо этого, если бы мы собирали статистику `SamplingStats`, то тогда смогли бы уже запустить эксперимент с большими размерами решетки. Однако гистограмма лучше иллюстрирует реальное распределение, которое само по себе может быть аппроксимировано гауссовским нормальным распределением, как вы можете знать.

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

```
import Simulation.Aivika.Trans
import Simulation.Aivika.Lattice
import Simulation.Aivika.Experiment.Histogram

meanUpTime = 1.0
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 1000.0,
                spcDT = 0.1,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation LIO Histogram
model =
  do let latticeDistribution :: Estimate LIO [Int]
         latticeDistribution =
           do k <- liftComp latticeMemberIndex
              return [k]

     let reduce :: [Int] -> [Int] -> Estimate LIO [Int]
         reduce ks1 ks2 = return $ ks1 ++ ks2

     let leaf = latticeDistribution

     ks <- foldEstimate reduce leaf

     runEstimateInStartTime $
       do xs <- ks
          let ys = fmap (fromIntegral) xs
              hs = histogram binScott [ys]
          return hs

main :: IO ()
main =
  do lat <- newRandomLattice 10
     hs <- runLIO lat $
           runSimulation model specs
     putStrLn "Histogram:"
     putStrLn (show hs)
```

Запустив эту модель в терминале, я получил следующий вывод[3].

[3] В 2024 году такая простая команда уже не сработает — лучше создать проект Cabal или Stack, как много раз упоминалось ранее в сносках.

```
$ runghc Distribution.hs
Histogram:
[(0.25,[1]),(1.25,[10]),(2.25,[45]),(3.25,[120]),(4.25,[210]),
 (5.25,[252]),(6.25,[210]),(7.25,[120]),(8.25,[45]),(9.25,[10]),
 (10.25,[1])]
```

Значения в квадратных скобках определяют вес соответствующих столбцов гистограммы. Даже несмотря на то, что индексы элемента целочисленны, гистограмма строится в предположении, что значения могут быть вещественными.

Теперь мы видим, что листовые завершающие узлы со средним значением индекса элемента используются примерно в 252 раза чаще, чем узлы с краевыми индексами элемента. Реальное значение достаточно условно, поскольку гистограмма была построена в предположении нецелочисленных чисел, но порядок величин может произвести впечатление. Так что, наш численный эксперимент вполне согласуется с теорией.

14.7 Критерий применимости

Поскольку узлы решетки могут иметь совершенно разную степень влияния на конечную оценку, как мы это увидели в разделе 14.6, то очень важно понимать то, когда решетка применима, а когда — нет. Это настолько краеугольный камень, что я посвятил отдельный раздел для такого замечания. Здесь легко допустить ошибку.

Вероятно, что этот тип вложенного моделирования применим к моделям ценообразования опционов из финансового моделирования. Например, мы могли бы расширить эти модели, введя туда элементы дискретно-событийного моделирования. Только остается вопросом то, есть ли реальная выгода от этого? Тем не менее, вы, как мой читатель, могли бы согласиться, что рассмотренный тип вложенного моделирования выглядит очень красивым, по крайней мере, с точки зрения теории.

14.8 Пример: модель ценообразования опционов

В качестве примера возьмем биномиальную модель ценообразования опционов. Она может быть легко реализована средствами Айвики на основе

вычисления LIO.

> Предположим, что опцион "пут"с ценой исполнения 110 долларов в настоящее время торгуется на уровне 100 долларов и истекает через год. Годовая безрисковая ставка составляет 5%. Ожидается, что цена будет увеличиваться на 20% и снижаться на 15% каждые шесть месяцев. Необходимо оценить цену опциона "пут".

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

import Simulation.Aivika.Trans
import Simulation.Aivika.Lattice
import Simulation.Aivika.Experiment.Histogram

-- размер решетки
n = 50

-- коэффициенты повышения и понижения
u0 = 1.2
d0 = 0.85

-- скорректированные коэффициенты с учетом размера решетки
u = exp (log u0 / (fromIntegral n / 2))
d = exp (log d0 / (fromIntegral n / 2))

-- начальная цена, по которой торгуется акция
s0 = 100.0

-- цена исполнения для опциона "пут"
strikePrice = 110.0

-- годовая безрисковая ставка
r = 0.05

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 1.0,
                spcDT = 0.1,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation LIO Double
model =
  do -- цена акции
     s <- newRef s0
```

```
    -- вычислить дерево изменения цен для акции
    runEventInStartTime $
      enqueueEventWithLatticeTimes $
      do k  <- liftComp latticeMemberIndex
         k0 <- liftComp latticeParentMemberIndex
         case k0 of
           Nothing -> return ()
           Just k0 | k == k0 ->
             modifyRef s (\x -> x * u)
           Just k0 | k == k0 + 1 ->
             modifyRef s (\x -> x * d)

    -- шаг времени для решетки
    dt <- liftParameter latticeTimeStep

    -- определить вероятность повышения
    let p = (exp (- r * dt) - d) / (u - d)

    -- оценить опцион в конечных точках времени
    let leaf :: Estimate LIO Double
        leaf =
          do x <- readObservable s
             -- поскольку это опцион "пут"
             return $ max (strikePrice - x) 0

    -- оценить опцион через предсказание
    let reduce :: Double -> Double -> Estimate LIO Double
        reduce x1 x2 =
          return $
          exp (- r * dt) * (p * x1 + (1 - p) * x2)

    price <- foldEstimate reduce leaf

    runEstimateInStartTime price

main :: IO ()
main =
  do lat <- newRandomLattice n
     e <- runLIO lat $
          runSimulation model specs
     putStrLn "Estimation:"
     putStrLn (show e)
```

После запуска в терминале я получил следующую оценку[4]:

```
$ runghc BinomialPricingModel.hs
Estimation:
14.153391452807556
```

Это пример не демонстрирует всех сильных сторон Айвики, поскольку он не включает в себя ни сложных событий, ни дискретных процессов. Вероятно, что вы сможете найти подходящее приложение для этого программного инструмента.

14.9 Заключение

Запуск вложенных дискретно-событийных моделей в узлах решетки может быть интересной областью для исследований и возможных приложений. Только следует быть очень осторожным, ясно осознавая, где этот тип вложенного моделирования применим, а где не применим.

[4] В 2024 году следует создать проект Cabal или Stack.

Приложение А

Установка Айвики

Инструкции ниже даны для случая использования Stack[1], хотя те же самые библиотеки могут быть установлены через Cabal. Обратите внимание на то, что использованные в инструкциях версии соответствуют времени написания этого документа. Актуальные версии могли измениться.

А.1 Использование библиотек только с открытым исходным кодом

При использовании библиотек только с открытым исходным кодом, вы можете добавить следующие зависимости в раздел `extra-deps` вашего файла *stack.yaml*:

```
- aivika-6.1
- aivika-transformers-6.1
- aivika-distributed-1.5
- aivika-experiment-5.4
- aivika-experiment-chart-5.4.1
- aivika-experiment-diagrams-5.4.1
- aivika-experiment-cairo-5.4.1
- aivika-realtime-0.4
- aivika-branches-0.4
- aivika-lattice-0.7
- aivika-gpss-0.7.1
```

Теперь эти библиотеки будут доступны в вашем проекте Stack.

[1] http://docs.haskellstack.org/

А.2 Использование пакетов расширения Айвики

Пакеты расширения Айвики включает в себя программные библиотеки, для которых существует своя собственная лицензия. Если кратко, то сейчас соответствующие пакеты бесплатны для образовательных и академических целей, для других же целей необходимо приобрести лицензию.

Сами пакеты расширения расположены по следующим адресам:

```
https://gitflic.ru/project/dsorokin/aivika-experiment-entity
https://gitflic.ru/project/dsorokin/aivika-experiment-entity-hdbc
https://gitflic.ru/project/dsorokin/aivika-experiment-provider
https://gitflic.ru/project/dsorokin/aivika-experiment-provider-distributed
https://gitflic.ru/project/dsorokin/aivika-experiment-report
```

А.3 Справочная документация по API

После установки Айвики вы можете собрать справочную документацию по API:

```
$ stack haddock
```

Приложение B

Интерфейс для графиков

Существует два интерфейса для графиков, которые применяются в Айвике через библиотеку `Chart`: на основе Cairo и на основе Diagrams. Оба взаимозаменяемы.

Первый интерфейс реализован в пакете `aivika-experiment-cairo`. Он требует того, чтобы на вашем компьютере была установлена библиотека Cairo. Я смог заставить ее работать только на системах Linux и macOS[1]. Наиболее простым способом установить будет через Cabal. Пришлось использовать пакет `gtk2hs-buildtools` для установки необходимых пакетов `cairo` и `Chart-cairo`. Я не тестировал этот интерфейс вместе со Stack.

Второй интерфейс реализован уже в пакете `aivika-experiment-diagrams`. Он работает на всех трех платформах: Windows, Linux и macOS. Его легко установить как с помощью Cabal, так и с помощью Stack.

Интерфейс на основе Cairo является самым быстрым. Он создает небольшие растровые файлы рисунков в формате PNG. Напротив, интерфейс на основе Diagrams создает векторные графические файлы в формате SVG. Файлы SVG более детализированные, но и они могут иметь временами довольно большой размер, что может привести к замедлению имитационного эксперимента. У меня нет личных предпочтений по тому, какой формат лучше выбрать.

[1] Давно не проверялось мною.

Приложение C

Трассировка имитации

После начала работы над новой моделью у вас может возникнуть желание проверить то, а ведет ли себя модель, как вы того ожидаете: в правильном ли порядке создаются события, работают ли процессы как должно и тому подобное. В таких случаях может быть полезным добавить трассировку имитационной модели. Для этого в Айвику добавлены очень простые комбинаторы, которые могут позволить вам создавать новые вычисления, которые бы уже показывали информацию о трассировке во время имитации.

Например, следующие комбинаторы трассируют моделирующие активности в рамках вычислений `Event` и `Process`:

```
traceEvent :: String -> Event a -> Event a
traceProcess :: String -> Process a -> Process a
```

Есть похожие трассирующие комбинаторы и для других моделирующих вычислений тоже. Здесь мы передаем отладочное сообщение и вычисление, трассировку которого мы бы хотели увидеть. Затем мы должны применить в нашей модели то вычисление, которое возвращают эти функции. Такое вычисление должно быть использовано вместо того исходного вычисления, которые мы передали вторым аргументом.

Для демонстрации подхода давайте возьмем нашу любимую модель из раздела 2.6 и слегка ее перепишем:

```
import Control.Monad.Trans

import Simulation.Aivika

meanUpTime = 1.0
```

```
meanRepairTime = 0.5

specs = Specs { spcStartTime = 0.0,
                spcStopTime = 10.0,
                spcDT = 1.0,
                spcMethod = RungeKutta4,
                spcGeneratorType = SimpleGenerator }

model :: Simulation Results
model =
  do totalUpTime <- newRef 0.0

     let machine :: Int -> Process ()
         machine i =
           do upTime <-
                traceProcess (show i ++ " has started working...") $
                randomExponentialProcess meanUpTime
              liftEvent $
                modifyRef totalUpTime (+ upTime)
              repairTime <-
                traceProcess (show i ++ " is to be repaired") $
                randomExponentialProcess meanRepairTime
              machine i

     runProcessInStartTime $ machine 1
     runProcessInStartTime $ machine 2

     let upTimeProp =
           do x <- readRef totalUpTime
              y <- liftDynamics time
              return $ x / (2 * y)

     return $
       results
       [resultSource
        "upTimeProp"
        "The long-run proportion of up time (~ 0.66)"
        upTimeProp]

main =
  printSimulationResultsInStopTime
  printResultSourceInEnglish
  model specs
```

При запуске этой имитации в редакторе Emacs я получил следующий вывод:

```
*Main> main
t = 0.0: 1 has started working...
t = 0.0: 2 has started working...
t = 0.4318062180996482: 2 is to be repaired
t = 0.5305397516818269: 2 has started working...
t = 0.5586381207729983: 1 is to be repaired
t = 0.7991693313890962: 1 has started working...
t = 1.1112112024351803: 1 is to be repaired
t = 1.6643746260347085: 2 is to be repaired
t = 1.856704114194633: 1 has started working...
t = 2.0962321743791383: 2 has started working...
t = 2.2388567478888683: 1 is to be repaired
t = 2.247968821638724: 1 has started working...
t = 2.2941823206736935: 1 is to be repaired
t = 2.531581157098128: 1 has started working...
t = 2.9454519613530263: 1 is to be repaired
t = 3.193197128356786: 1 has started working...
t = 3.300465964786781: 1 is to be repaired
t = 4.187174883320544: 2 is to be repaired
t = 4.256014384274227: 2 has started working...
t = 4.358482637694295: 1 has started working...
t = 4.75598928287998: 2 is to be repaired
t = 5.103523956621543: 1 is to be repaired
t = 5.174350666896869: 2 has started working...
t = 6.826423919789626: 1 has started working...
t = 7.5805949343846315: 2 is to be repaired
t = 7.996224752680289: 2 has started working...
t = 8.267767012096208: 2 is to be repaired
t = 8.58683886539276: 2 has started working...
t = 8.638961434698468: 1 is to be repaired
t = 8.69742530559413: 1 has started working...
t = 9.173995410698865: 2 is to be repaired
t = 9.264242025802165: 2 has started working...
t = 9.532011600233057: 1 is to be repaired
----------

-- simulation time
t = 10.0

-- The long-run proportion of up time (~ 0.66)
upTimeProp = 0.6316926332958837
```

Безусловно, что выбранное конечное время моделирования слишком мало для хорошей оценки результата, но здесь благодаря отладочным сообщениям мы видим, как ведет себя модель в соответствии с нашими ожиданиями.

Литература

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., USA, 1985.

[2] Samadi B. *Distributed simulation, algorithms and performance analysis*. PhD thesis, University of California, Los Angeles, 1985.

[3] Jefferson D.R. and B. Beckman et al. The Time Warp operating systems. *11th Symposium on Operating Systems Principles*, 21:77–93, 1987.

[4] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

[5] John Hughes. Programming with arrows. In *Advanced Functional Programming*, pages 73–129, 2004.

[6] `iThink` Software. `http://www.iseesystems.com`, 2014. Accessed: 1-May-2014.

[7] Robert Macey and George Oster. `Berkeley Madonna` Software. `http://www.berkeleymadonna.com`, 2014. Accessed: 1-May-2014.

[8] Norm Matloff. Introduction to discrete-event simulation and the `SimPy` language. `http://simpy.readthedocs.org/en/latest/`, 2008. Accessed: 1-May-2014.

[9] Henrik Nilsson and Antony Courtney et al. `Yampa` Library, Version 0.9.5. `http://hackage.haskell.org/package/Yampa`, 2014. Accessed: 1-May-2014.

[10] Ross Paterson. A new notation for Arrows. In *In International Conference on Functional Programming*, ICFP '01, pages 229–240. ACM, 2001.

[11] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: An adventure in financial engineering (functional pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 280–292, New York, NY, USA, 2000. ACM.

[12] 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.

[13] Fujimoto R.M. *Parallel and Distributed Simulation Systems*. Wiley Interscience, 2000.

[14] Thomas Schriber. *Simulation using GPSS*. Wiley, 1974.

[15] `SimPy` Library. `http://simpy.readthedocs.org/en/latest/`, 2014. Accessed: 1-May-2014.

[16] `AnyLogic` Software. `http://www.anylogic.com`, 2014. Accessed: 1-May-2014.

[17] Ilya I. Trub. *An Object-oriented Modeling in C++*. Piter, Russia, 2006. (In Russian).

[18] `Vensim` Software. `http://vensim.com`, 2013. Accessed: 1-May-2014.

Предметный указатель