DOI: 10.24411/1993-8314-2019-10028

***Д. Э. Сорокин**, магистр прикладной математики и информатики, г. Йошкар-Ола, david.sorokin@gmail.com*

Распределенное имитационное моделирование с Aivika

В статье представлена система имитационного моделирования Айвика (*англ.* Aivika), где особое внимание уделяется модулю распределенного моделирования, реализующего оптимистичный метод деформации времени. Показано, как в распределенной имитации можно использовать основные парадигмы дискретно-событийного моделирования, в частности, процесс-ориентированную парадигму. Также представлено, как в такой имитации можно использовать GPSS-подобный предметно-ориентированный язык. Описаны важные детали реализации. В конце приведены результаты тестирования распределенного модуля системы Айвики.

Ключевые слова: дискретно-событийное моделирование, распределенное моделирование, оптимистичный метод деформации времени, язык моделирования GPSS, дискретные процессы

Введение

Имитационное моделирование является одним из важных инструментов изучения поведения сложных систем, таких как телекоммуникационные сети, производственные процессы или трафик движения городского автотранспорта. Часто используются методы, нацеленные на последовательную имитацию, но с увеличением размерности задачи возникает желание распределить имитационный эксперимент сразу по нескольким *логическим процессам* так, чтобы исполнение разных частей имитационной модели шло параллельно, но согласованно при этом. Перед *аналитическими* методами распределенного моделирования [9] ставится задача, чтобы они возвращали результаты, эквивалентные тому, что возвращают методы последовательной имитации. Более того, некоторые задачи не умещаются в память одного компьютера. Для них просто нет альтернативы распределенному моделированию.

В настоящей работе описана созданная автором система имитационного моделирования Айвика [5, 19]. Показано, как на основе единого подхода мы можем запускать как последовательную [14], так и распределенную имитацию [15]. Поддерживаются основные парадигмы дискретно-событийного моделирования, включая процесс-ориентированную парадигму. Помимо этого, реализован GPSS-подобный [16] встраиваемый предметно-ориентированный язык, который хотя полностью не эквивалентен оригинальному языку GPSS, но в некоторых случаях возвращает те же результаты. Также поддерживаются обработчики таймера и тайм-аута для агентов.

Это буквально означает, что мы можем использовать модели с дискретными событиями и процессами, с цепочками блоков, очередями, с вытеснением прибора, с многоканальными устройствами, а также с агентами, и все это будет одинаково работать как в случае последовательной, так и распределенной имитации. При этом Айвика поддерживает *оптимистичный* метод *деформации времени* [10] (*англ.* Time Warp), который известен сложностью своей реализации. Подобные ре-

зультаты автор работы не встречал ранее в литературе.

Также в работе рассматривается одна модель замкнутой сети очередей [18], которая использовалась для тестирования модуля распределенного моделирования в Айвике.

Здесь стоит заметить, что система Айвика распространяется в открытых кодах по разрешительной лицензии BSD3 через серверы Hackage DB, что позволяет практически без ограничений и бесплатно использовать ее как в академических, так и коммерческих проектах.

Существующие системы распределенного моделирования

Если говорить об аналитических методах распределенного моделирования, то основная проблема связана с возможным возникновением *парадокса времени* [2]. При этом логическому процессу приходит от другого логического процесса сообщение, которое нужно было бы обработать еще в прошлом состоянии модели. У каждого сообщения есть метка, которая задает модельное время, в которое сообщение должно быть обработано. Аналитические методы гарантируют, что сообщения обрабатываются упорядоченно по модельному времени. В настоящей работе рассматриваются только аналитические методы моделирования.

Консервативные методы распределенного моделирования исключают саму возможность возникновения парадокса. Например, в алгоритме *пустых сообщений* [7] (*англ.* Null Message Algorithm) рассылаются, помимо самих сообщений, также пустые сообщения, которые информируют другие логические процессы о текущем локальном модельном времени логического процесса. На уровне модели вводится обязательный параметр «предсказания» (*англ.* lookahead), который, имея положительное значение, задает минимальное время доставки сообщений. Все это вместе позволяет определить, насколько далеко каждый логический процесс может уйти во время имитации, будучи ограниченным этим самым параметром предсказания. Очевидно, что данный параметр часто имеет решающее значение для эффективности распределенной имитации. Это — одно из слабых мест консервативных методов.

Напротив, оптимистичные методы вполне допускают возникновение парадокса времени, но в таком случае они запускают некоторую восстанавливающую процедуру, которая возвращает модель в прошлое состояние с тем, чтобы мы могли обработать, наконец, проблемное входящее сообщение. В случае алгоритма деформации времени восстановление прошлого модели происходит за счет откатов, которые могут быть каскадными, охватывая сразу множество логических процессов. Айвика реализует именно данный алгоритм деформации времени.

Откаты подразумевают или ведение журнала операций или чего-то похожего по сути, например, журнала точек восстановления состояний. В Айвике создается журнал операций, которые могут быть обращены, чтобы получить прошлое состояние модели. В любом случае, возникает журнал, который разрастается в объеме. К счастью, мы можем сократить размер такого журнала, если нам известно *глобальное виртуальное время*, которое задает нижнюю границу по всем значениям локального модельного времени для всех логических процессов.

Такая очистка журнала от прошлых данных сравнима чем-то со сборкой мусора в языках программирования с автоматической системой управления памятью, таких как Java или C#.

Для вычисления глобального виртуального времени может быть использован *алгоритм Самади* [12], который и реализован в Айвике. Здесь логические процессы общаются с некоторым сервером времени, от которого получают текущее значение оценки глобального времени, а также передают ему свое локальное модельное время. Суть ал-

горитма в том, что вводятся отчеты о доставке, и если заданный логический процесс уже передал серверу времени свое локальное значение, то вплоть до следующего сеанса синхронизации с сервером времени, логический процесс помечает все исходящие отчеты о доставке для других логических процессов. Это нужно для того, чтобы те процессы в свою очередь учли время получения сообщений при своих сеансах синхронизации с сервером. Также учитывается время получения тех сообщений, на которые еще не пришел отчет о доставке.

Главным недостатком алгоритма деформации времени является его же достоинство — откаты. Если откатов слишком много, то имитация существенно замедляется. Поэтому часто вводят искусственное ограничение в виде *горизонта моделирования* [9], который определяет то, как далеко может уйти логический процесс во время имитации относительно глобального виртуального времени. Возникает другой парадокс, когда ограничивая скорость каждого логического процесса в отдельности, мы можем иногда увеличить совокупную скорость всей распределенной имитации.

Что касается реализаций, то в нашей стране ведутся работы по созданию таких распределенных имитационных систем как Мера [3], Диана [4] и Triad.Net [1]. Примерами зарубежных систем являются SPEEDES [20], WarpIV [21, 22] и OMNeT++ [13, 11]. В некоторых из этих систем реализованы как консервативный, так и оптимистичный методы одновременно. Более того, многие реализуют стандарт HLA [2], который призван интегрировать между собой имитационные модели, разработанные с помощью разных систем моделирования.

Многие из вышеуказанных систем нашли промышленное применение.

Например, отечественная система Мера имеет ядро, которое является процессно-ориентированной системой дискретного имитационного моделирования, где взаимодействие процессов осуществляется с помощью механизма передачи сообщений. Реализованы как консервативный, так и оптимистичный методы синхронизации модельного времени. Поддерживается архитектура HLA. Система Мера была использована при разработке Автоматизированной системы управления технологическими процессами Северомуйского тоннеля.

Система Диана использует UML-представление, которое транслируется в код на C++. Система совместима со стандартом HLA. Диана была апробирована при моделировании функционирования бортовых вычислительных систем морских навигационных комплексов.

Система Triad.Net является событийно-ориентированной системой имитационного моделирования. Основным способом моделирования является описание модели с помощью языка Triad. Предусмотрена трансляция такого описания специальным компилятором в объектный код для эффективного исполнения.

По поводу SPEEDES и более новой системы WarpIV можно предполагать, что они нашли применение в американской оборонке. Так, система WarpIV реализует и консервативный, и оптимистичный методы. Ориентирована она на использование языка C++. Есть поддержка HLA.

Касательно системы OMNeT++, ее исходный код доступен широкой аудитории по собственнической лицензии. Для коммерческого использования необходимо приобрести лицензию. Ядро системы использует язык C++. Существует поддержка консервативного метода распределенного моделирования. Также используется специальный высокоуровневый язык NED для описания моделей. Предлагается визуальный редактор для построения таких моделей. Судя по всему, система OMNeT++ популярна для моделирования телекоммуникационных сетей.

Единый подход к моделированию в Айвике

Ниже представлен единый подход к моделированию, который используется в Айвике как для последовательной имитации, так и для распределенной. Более того, этот подход применим и для вложенного моделирования [19]. Это во многом стало возможным благодаря выбранной платформе программирования.

Язык программирования Haskell

Айвика немного не похожа на другие системы моделирования, которые больше опираются на использование императивных языков программирования, таких как C++ и Java. Айвика же по всей своей сути основана на идеях *функционального программирования*.

Если императивную программу можно рассматривать как пошаговую инструкцию действий, то программу на функциональном языке, пусть это звучит как тавтология, можно рассматривать как вычисление некоторой функции. Сложные функции создаются через композицию более простых, а для того, чтобы было легче судить о свойствах функций, добавляют требование об отсутствии неявных побочных эффектов, что позволяет сделать язык программирования *ссылочно-прозрачным*, а сами побочные эффекты могут быть явно выражены при этом в системе типов. Примером такого языка является язык *чистого функционального программирования* Haskell, на котором и написана система моделирования Айвика.

Замечено, что программы на функциональных языках часто выглядят декларативно. Поэтому неудивительно, что модели, записанные с помощью Айвики, могут напоминать высокоуровневые языки моделирования, такие как SIMSCRIPT или GPSS. Айвика по сути превращает общецелевой язык программирования Haskell и в язык моделирования тоже.

Для Haskell создан компилятор GHC, который умеет оптимизировать код, удаляя при компиляции лишние абстракции, которые непременно возникают при такой трактовке побочных эффектов, а они необходимы для проведения имитационных экспериментов.

Более того, тотальная ссылочная прозрачность дает преимущество сборщику мусора, поскольку старые объекты не могут ссылаться на новые, если это не явные ссылки, которых обычно мало в программе. Это позволяет исполняющей системе GHC эффективно обрабатывать кратко-живущие объекты, которые могут создаваться в значительных количествах.

Еще такой важный момент, что практически вся инфраструктура для Haskell, включая компилятор GHC и основные библиотеки, распространяется в открытых кодах, что иногда имеет большое значение.

Если Haskell долгое время оставался чисто академической разработкой, то последние годы его все чаще используют в промышленном программировании. Например, Haskell используют в американской компании Facebook.

Событийно-ориентированная парадигма

В Айвике процесс-ориентированная парадигма дискретно-событийного моделирования неявно сводится к событийно-ориентированной парадигме. Каждый запуск имитации имеет очередь событий. Создаются обработчики событий, которые должны активироваться в заданное модельное время, то есть, обработчики упорядочены по времени. Во время активации обработчика события можно создавать новые события.

Для передачи в обработчики данных и соответствующего контекста используются за-

мыкания, а сами обработчики дискретных событий в случае последовательной имитации представлены вычислением Event, которое имеет следующее определение, как показано в листинге 1.

По сути это функция от точки модельного времени Point, которая возвращает вычисление IO некоторого значения переменного типа a. Здесь IO — это стандартная монада библиотеки языка Haskell, которую часто называют еще императивной. Монада является частным случаем концепции вычисления. Вышеприведенный тип Event тоже является монадой. Если говорить в контексте более привычных языков программирования Java и C++, то их программы можно рассматривать как неявные вычисления IO. Внутри такого вычисления можно создавать ссылки, изменяемые массивы и менять их состояния. Сами же типы IO и Event чем-то похожи на шаблоны C++ или generics из языка Java.

Точка модельного времени Point содержит ссылку на очередь событий. Так мы можем создавать новые события в рамках вычисления текущего обработчика события, что отражено в листинге 2.

Здесь мы передаем в рамках вычисления Event первым аргументом новое время активации и вторым аргументом соответствующий обработчик события. Стандартный тип () является особым. Он удобен для выражения побочных эффектов. Создание нового события по сути и есть побочный эффект в рамках текущего вычисления. Здесь можно трактовать тип () как некоторую версию типа void из языков C++ и Java, но еще и наделенную тем свойством, что у этого типа может быть значение, которое обозначается тоже как ().

Если говорить своими словами, то монада — это такое общее свойство некоторых параметрических типов, где мы можем создавать новые вычисления из примитива и объединять вычисления в составные, где продолжение вычисления может зависеть от предыдущего вычисления.

Это настолько общее свойство, что в нашем случае позволяет запрограммировать широкий спектр моделируемых активностей. Причем, вычисление обработчиков дискретных событий Event становится *композиционным*. Это значит, что мы можем использовать вычисления как строительные блоки, создавая сложные вычисления из более простых вычислений, что и будет продемонстрировано далее при рассмотрении процесс-ориентированной парадигмы дискретно-событийного моделирования.

Более того, в языке Haskell поддерживается «нотация-*do*», которая значительно упрощает создание монадических вычислений, таких как IO или Event. Тогда программный код становится внешне похожим на код на обычных императивных языках, таких как Java или C++. В нашем же случае это будет означать, что код в случае использования дискретных процессов будет похож на модель, написанную, например, на языке моделирования SIMSCRIPT, но об этом чуть далее.

Также здесь заметим, что в рамках вычисления Event можно определять обработ-

Листинг 1. Определение вычисления Event
Listing 1. Event computation definition

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

Листинг 2. Сигнатура функции создания дискретного события
Listing 2. A signature of the function that allows creating discrete events

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

чики таймера и таймаута для агентов, что позволяет в Айвике комбинировать дискретно-событийное моделирование с элементами агентного моделирования. Еще заметим, что время-ориентированная парадигма сводится к событийно-ориентированной, если постоянно запускать обработчики событий с некоторым маленьким временным шагом через очередь событий.

Процесс-ориентированная парадигма

Функциональное программирование раскрывается в полной мере, когда мы определяем дискретные процессы через обработчики дискретных событий. Как было замечено ранее, последние обладают свойством композиционности, что и используется в следующем основанном на продолжениях вычислении Cont, как показано в листинге 3.

Это уже могло бы быть дискретным процессом, поскольку обладает тем уникальным свойством, что такое вычисление можно приостановить, а потом продолжить в любой другой момент модельного времени через очередь событий, используя введенную ранее функцию enqueueEvent, чтобы сымитировать задержку на заданный промежуток модельного времени.

Здесь функции внутри вычисления Cont передается другая функция, которая и известна в литературе как продолжение. На самом деле, в Айвике тип Cont устроен чуточку сложнее, и в нем целых три продолжения вместо одного: (1) для основного вычисления; (2) для обработки исключительных ситуаций; (3) для моделирования экстренной отмены вычисления.

Отмена требует использования изменяемой ссылки. В Айвике здесь еще используются ссылки для моделирования вытеснения процесса, например, что происходит при вытеснении прибора GPSS. Только вытеснение процесса — это уровень непосредственной реализации, а вытеснение прибора — это уже уровень прикладного API для моделирования. В целом же, использование побочных эффектов внутри Cont этим и ограничивается. Что особенно важно, при этом вычисление Cont сохраняет свойство композиционности. Тип Cont также является монадой.

Помимо имитации задержки по модельному времени, иногда бывает нужно усыпить дискретный процесс на неопределенное время или немедленно прервать дискретный процесс, пока он придерживался через очередь событий. Тогда вот это продолжение () — > Event () нужно где-то хранить. Для этого вводится тип идентификаторов дискретного процесса ProcessId, который становится хранилищем подобной информации, связанной с текущим процессом.

Тогда вычисление самого дискретного процесса получает следующее простое выражение, что можно увидеть в листинге 4.

Этот тип по-прежнему сохраняет свойство композиционности, и он также является монадой, что позволяет использовать нотацию-*do* для создания вычислений Process.

Листинг 3. Определение вычисления Cont
Listing 3. Cont computation definition

```
newtype Cont a = Cont ((a -> Event ()) -> Event ())
```

Листинг 4. Определение вычисления Process
Listing 4. Process computation definition

```
newtype Process a = Process (ProcessId -> Cont a)
```

Перед тем, как показать небольшой пример определения дискретного процесса, введем собственно функцию, которая придерживает дискретный процесс на заданный промежуток модельного времени. Функция показана в листинге 5.

В обычных моделях удобнее использовать вспомогательные функции, которые сначала порождают некоторое значение для случайного промежутка по заданному вероятностному распределению, а потом уже вызывают функцию holdProcess. Результат функции holdProcess является вычислением Process, которое обладает свойством композиционности, а значит, результирующее вычисление можно использовать как строительный блок при создании более сложных вычислений.

Для примера возьмем модель станка, который периодически ломается. Его активность может быть выражена в Айвике следующим образом, что отражено в листинге 6.

Мы сначала моделируем наработку станком в течение промежутка модельного времени upTime. Функция randomExponentialProcess как раз является вспомогательной. Она порождает некоторое случайное значение по показательному распределению с заданным средним, а потом вызывает функцию holdProcess с этим значением, используя последнее как промежуток модельного времени для задержки. Дальше в функции modifyRef идет изменение ссылки, но поскольку такое действие происходит в рамках вычисления Event, то мы действие должны преобразовать к вычислению Process, что и делает здесь функция liftEvent. После этого мы моделируем поломку станка в течение промежутка времени repairTime, а затем повторяем все действия рекурсивно, пока не наступит конечная точка моделирования.

Заметим, что модельное время здесь меняется скачками, но сам код выглядит так, как будто исполнение происходит линейно. Именно это и привлекает многих исследователей в таких специализированных языках моделирования как SIMSCRIPT. Только здесь Haskell, общецелевой язык программирования как Java или C++, где есть массивы, словари, ввод-вывод в файлы, и многое другое.

Дискретные процессы естественным образом интегрируются с событиями. В рамках обработчиков событий можно запускать новые процессы, а процессы могут выполнять код обработчиков. То же относится к обработчикам таймера и таймаута для агентов. Это является прямым следствием данного выше определения вычисления Process.

Листинг 5. Сигнатура функции задержки дискретного процесса
Listing 5. A signature of the function that holds the discontinuous process

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

Листинг 6. Модель периодически ломающегося станка после наработки
Listing 6. A model of the machine that has to be repaired after random up time

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

Здесь читатель может задаться вопросом, а какое это имеет отношение к распределенному моделированию? Все просто. Этот же самый код работает и в случае оптимистичного метода деформации времени распределенного дискретно-событийного моделирования, только сигнатура типа будет немного другой, но обо всем по порядку.

GPSS-подобный язык

Используя свойство композиционности вычислений дискретного процесса Process, мы можем составить новый тип вычислений Block для выражения конструкций, напоминающих блоки языка моделирования GPSS. Это можно увидеть в листинге 7.

Как мы увидим ниже, с помощью таких вычислений можно программировать имитационные модели на GPSS-подобном языке, который поддерживает очереди, захват и вытеснение прибора, использование многоканальных устройств и многое другое из оригинального языка моделирования GPSS.

Здесь же заметим, что вычисление Block на самом деле является частным случаем стрелки Клейсли, параметризованной по монаде Process. Это вычисление также обладает свойством композиционности, только это уже не монада, а стрелка. Цепочка блоков может быть скомпонована в новый блок. При этом поддерживаются безусловные переходы между блоками, а терминальный блок будет иметь тип `Block a ()`.

Для запуска модели, составленной из вычислений Block, нам еще понадобится блок генератора, который по заданной терминальной цепочке блоков будет создавать транзакты и затем пропускать их через эту цепочку, моделируя желаемую активность. Для этого в Айвике вводится тип GeneratorBlock, согласно листингу 8.

Для примера возьмем следующую простейшую модель GPSS с одной очередью и одним прибором, как показано в листинге 9.

Без блока генератора ее основная терминальная цепочка блоков в Айвике выглядит так, как отражено в листинге 10.

Листинг 7. Определение вычисления Block
Listing 7. Block computation definition

```
newtype Block a b = Block { blockProcess :: a -> Process b }
```

Листинг 8. Определение вычисления GeneratorBlock
Listing 8. GeneratorBlock computation definition

```
newtype GeneratorBlock a =
  GeneratorBlock { runGeneratorBlock :: Block a () -> Process () }
```

Листинг 9. Простейшая имитационная модель GPSS
Listing 9. A simple GPSS simulation model

```
GENERATE 18,6
QUEUE JOEQ
SEIZE JOE
DEPART JOEQ
ADVANCE 16,4
RELEASE JOE
TERMINATE
```

Здесь Айвика вернет ту же статистику по очереди и прибору, что возвращает симулятор GPSS. Расхождение возможно в том случае, если мы начинаем опираться в модели на приоритеты транзактов при активации следующего блока, жестко полагаясь на строгую детерминированную последовательность активации. В Айвике блоки активируются не в таком строгом порядке, но внутри блоков приоритеты транзактов принимаются во внимание по правилам языка GPSS, используя соответствующую стратегию очереди там, где необходимо.

Очевидно, что вычисления Process и Block эквивалентны. Поэтому в рамках GPSS-подобной модели мы можем интегрировать блоки с дискретными процессами, дискретными событиями и агентами.

Хорошая новость состоит в том, что это все работает и в случае оптимистичного метода распределенного моделирования, но прежде, чем перейти к этой теме, нам нужно сделать небольшой логический шаг, который будет иметь важные последствия, тем не менее.

Обобщение вычислений

Выше были приведены моделирующие вычисления для случая последовательной имитации. Как было сказано выше, все началось с вычисления обработчика дискретного события Event, где использовалась императивная монада IO.

Теперь обобщим это вычисление на случай произвольной монады, которую сделаем параметром типа. Здесь и далее вводятся новые типы, которые для удобства чтения и написания программ имеют в Айвике те же самые названия, но располагаются они уже в других модулях. Так, новое обобщенное вычисление приведено в листинге 11.

Здесь переменный тип m задает некоторую монаду, по которой мы фактически параметризуем систему моделирования Айвика. В самом типе это не отражено, но практически все функции Айвики явно в своих сигнатурах накладывают следующие ограничения на то, какой должна быть переменная монада m:

1) в рамках вычисления Event m мы должны уметь создавать новые события с помощью функции аналогичной enqueueEvent, которая была приведена ранее;

2) мы должны уметь создавать ссылки и менять их состояние в рамках того же вычисления Event m;

3) Айвика еще накладывает условие, что вычисление m должно поддерживать обработку исключительных ситуаций, но это не так принципиально для самого метода.

Листинг 10. Терминальная цепочка блоков GPSS в Айвике
Listing 10. The terminating chain of GPSS blocks in Aivika

```
let chain =
      queueBlock joeq 1 >>>
      seizeBlock joe >>>
      departBlock joeq 1 >>>
      advanceBlock (randomUniformProcess_ (16 - 4) (16 + 4)) >>>
      releaseBlock joe >>>
      terminateBlock
```

Листинг 11. Обобщенное вычисление Event
Listing 11. Generalized Event computation

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

То есть, если некоторое вычисление m удовлетворяет этому контракту, то взамен Айвика предоставляет [17] общецелевую систему дискретно-событийного моделирования с дискретными процессами Process m и GPSS-подобными блоками Block m, которые тоже параметризуются по этому переменному вычислению. При этом исходный код последовательной и обобщенной версий Айвики очень похожи, отличаясь, главным образом, в сигнатурах функций и некоторых незначительных деталях.

Реализация распределенного моделирования

Теперь мы можем перейти к тому, как устроено распределенное моделирование в Айвике. Если кратко, то для поддержки распределенного моделирования нам нужно предоставить некоторое вычисление, которое бы удовлетворяло контракту обобщенной версии Айвики. В текущей реализации есть одно такое вычисление DIO, которое соответствует названному контракту, и которое поддерживает оптимистичный алгоритм деформации времени с журналом откатов.

Ссылка при изменении своего состояния пишет операцию восстановления старого значения в журнал откатов. Сам журнал состоит из небольших блоков массивов, состоящих из 512 элементов и соединенных в двусвязный список. В массивах упорядоченно хранятся операции с соответствующими временными метками. Структура данных оптимизирована для добавления элементов в один конец и удаления с другого конца.

В Айвике дискретные события и асинхронные сообщения являются разными сущностями. Каждому входящему сообщению соответствует некоторое дискретное событие, но не наоборот.

Очередь событий является краеугольным камнем всей реализации в Айвике. Если в последовательной версии очередь событий реализована как императивная двоичная куча на основе массивов, то в случае распределенной имитации используется *чистая функциональная структура данных*, помещенная в изменяемую ссылку. При каждом изменении очереди, будь то добавление нового события или изъятие с последующей активацией события с минимальным временем, происходит сохранение старой версии очереди событий в журнале откатов. Это позволяет относительно быстро откатывать прошлое состояние очереди событий при сравнительно небольших накладных расходах на добавление и обработку событий. Причем метод хорошо масштабируется и способен эффективно обработать даже большое число одновременных событий, что мы увидим далее при тестировании.

Спецификой вычисления DIO является то, что для него уже определяются функции по обмену асинхронными сообщениями между логическими процессами: отдельно для отправки и приема сообщений. По очевидным причинам такие сообщения должны быть неизменяемыми, и это вполне вписывается в общую философию функционального программирования. Если отправка сообщения осуществляется в рамках вычисления Event, то приём в модель входящих сообщений происходит через производное вычисление Signal, которое напоминает интерфейс IObservable из платформы программирования .NET. Для краткости это вычисление Signal опущено. Оно удобно для обработки внешних событий.

Разделение сообщений и дискретных событий позволяет поддерживать обычные приемы дискретно-событийного моделирования в рамках реализации оптимистичного алгоритма распределенного моделирования, поскольку обработка событий относительно быстрая, а вот сообщения по своей природе несут некоторые накладные расходы, не говоря о том, что они могут вызывать откаты в самой модели. Дискретные же события никаких откатов не вызывают сами по себе. Они

только могут привести к отсылке сообщений, а те в свою очередь могут вызвать и откат.

Также стоит особо отметить, что оптимистичный метод моделирования хорошо согласовывается с функциональным программированием, где обычной практикой является подход, когда некий объект реализуется как чистая функциональная структура данных, а потом помещается в изменяемую ссылку. В таком случае объекты откатываются целиком, а количество записей в журнале откатов сокращается. Между тем, чистые функциональные структуры данных полезны также для представления транзактов и в последовательной имитации.

В распределенном модуле Айвики реализована особая поддержка для операций ввода-вывода, которые представляют собой определенную сложность, поскольку такие операции не могут быть обращены в общем случае. Существует защитный механизм от перегрузки событиями или сообщениями. Также реализован режим имитации устойчивой к временным разрывам связи [6].

Наконец, Айвика использует Cloud Haskell [8] для непосредственной передачи сообщений между логическими процессами. Поэтому логические процессы могут быть запущены и как отдельные потоки в рамках одного системного процесса, и как отдельные системные процессы на одном компьютере, и как процессы на разных компьютерах, в любом сочетании.

Тестовая модель

Для тестирования использовалась следующая модель замкнутой сети очередей, которая обычно применяется для оценки эффективности консервативных методов распределенной имитации [13]:

Модель состоит из N тандемов очередей, где каждый тандем состоит из переключателя и k очередей серверов с показательным временем обработки транзактов. Последние очереди завернуты в цикл к своим переключателям. Каждый переключатель, используя равномерное распределение, случайно выбирает первую очередь одного из тандемов для передачи транзакта. Очереди и переключатели соединены так, что время передачи транзакта имеет везде положительные значения для времени задержки.

Модель была запрограммирована средствами Айвики. В свободный доступ на GitHub выложена соответствующая реализация [18]. Для ее запуска не требуется знаний языка Haskell. Нужно лишь установить средство сборки Stack на компьютер или на все те компьютеры, где предполагается запустить распределенную имитацию.

Ниже приведены оценки эффективности модуля распределенного моделирования, сделанные на основе экспериментов, проведенных на одном компьютере с процессором Intel Core i7 CPU 920@2.67ГГц x 8, у которого 4 ядра с поддержкой технологии Hyper-threading, то есть, компьютер имеет условно 8 виртуальных ядер. Два ядра предоставлялись системе исполнения GHC, а значит, можно было запустить не более 5-6 параллельных логических процессов. Как мы увидим ниже, в одном случае 6 логических процессов было даже много.

Безусловно, если запускать модель на кластере, то скорость может упасть, в том числе по той причине, что увеличится число откатов в имитации из-за задержек во время передачи сообщений по сети. Такие запуски могут быть особо подвержены влиянию горизонта моделирования. Более того, каждому логическому процессу нужно будет тогда выделить дополнительное ядро процессора для осуществления непосредственной передачи и приема сообщений. Однако скорость распределенной имитации на кластере можно увеличить, если заменить равномерное распределение тандемов на другое, отдавая большее предпочтение текущему тандему при переключении транзактов. Тогда уменьшится число асинхронных сообщений между логическими процессами. Тестовая модель содер-

жит соответствующий параметр в конфигурационном файле.

Модель имеет параметр L, который задает задержку доставки транзакта от переключателя к первой очереди выбранного тандема. Для консервативных методов это будет собственно сам параметр предсказания, от которого они сильно зависят. В оптимистичном методе задержка может быть любой, но чтобы увидеть зависимость, мы также полагаем L константным, а зависимость, безусловно, есть, но она имеет другую природу. Например, L=1 будет давать заметное количество откатов, тогда как при L=500 откатов будет меньше.

В качестве начальных условий количество серверов в тандеме было задано как k=50, и в каждый тандем в начале имитации было загружено 100 транзактов.

Время обработки сервером считалось по показательному распределению со средним 10, а также добавлялась 1 единица модельного времени для пересылки транзакта на следующую очередь или переключатель. Начальное модельное время полагалось равным 0, а конечное было задано как 100000 (сто тысяч).

Количество тандемов N было переменным.

Имитация запускалась в разных режимах. Измерялось соответствующее время моделирования:

- *T_seq* соответствует режиму, когда все тандемы очередей моделируются в рамках одной последовательной имитации, запущенной с помощью последовательного модуля Айвики;
- *T_pseudo_seq* соответствует режиму, где все тандемы очередей также моделируются в рамках одной последовательной имитации, но уже запущенной на одном логическом процессе распределенного модуля Айвики без какого-либо обмена сообщениями;
- *T_par* соответствует режиму, когда каждый тандем моделируется отдельным логическим процессом, но все логические процессы запускаются в рамках одного системного процесса на одном компьютере, используя многопоточность.

Особый интерес представляют отношения этих величин относительно друг друга.

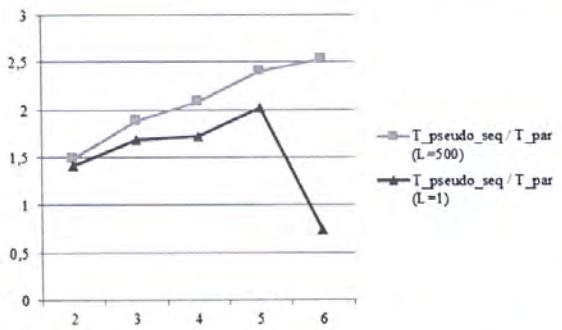

Рис. 1. Прирост от использования параллельных логических процессов для моделирования тандемов очередей по сравнению с последовательной моделью (ось OY) относительно соответствующего количества тандемов очередей N (ось OX)

Fig. 1. The speed-up ratio when using separate parallel logical processes for modeling each queue tandem in comparison with the sequential model (OY axis) relative to the corresponding number N of queue tandems (OX axis)

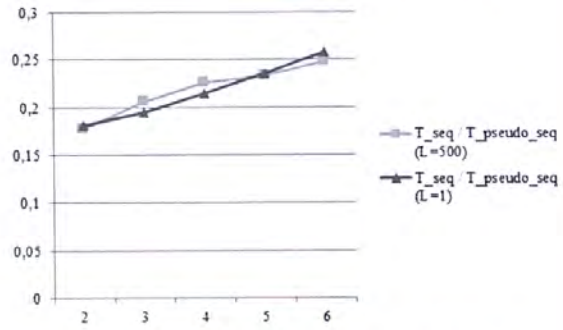

Рис. 2. Оценка эффективности одного логического процесса распределенного модуля без учета обмена сообщениями по сравнению с последовательным модулем (ось OY) относительно количества тандемов очередей N (ось OX)

Fig. 2. Estimation for the efficiency of single logical process of the distributed module in comparison with the sequential module (OY axis) relative to number N of queue tandems (OX axis) when there is no message passing

Масштабирование распределенного модуля

Прирост от использования параллельности можно увидеть на рисунке 1. Мы видим, что модуль распределенного моделирования масштабируется в общем случае. Однако в одном случае при использовании 6 логических процессов и маленького значения параметра L происходит снижение скорости имитации. Это можно объяснить тем, что возникает недостаток в вычислительных мощностях, и логические процессы перестают успевать друг за другом, что ведет к увеличению числа откатов в имитации.

Эффективность распределенного модуля вне контекста передачи сообщений

Также интересно знать, насколько велики накладные расходы от того, что мы добавляем саму возможность проведения распределенной имитации, например, каков эффект от неуклонного ведения журнала откатов и использования чистой функциональной структуры данных для реализации очереди событий?

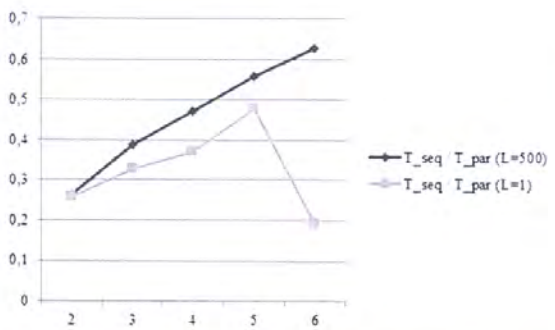

Рис. 3. Реальный эффект от распараллеливания имитационной модели (ось OY) относительно количества тандемов очередей N (ось OX))

Fig. 3. The actual effect of parallelizing the simulation model (OY axis) relative to number N of queue tandems (OX axis)

На рисунке 2 показано, во сколько раз распределенный модуль медленнее последовательного, когда есть только один логический процесс и нет никакого обмена сообщениями. Здесь получается, что он медленнее примерно в 4-5 раз, но другие замеры показывают, что он может быть медленнее и в 8-15 раз на простых моделях, где последовательный модуль быстро разгоняется. По-видимому, разрыв в скорости уменьшается при увеличении числа одновременно обрабатываемых дискретных событий, что и демонстрирует график.

Есть ли выигрыш от распараллеливания задачи?

Наконец, зададимся вопросом, а можно ли получить выигрыш от распараллеливания модельной задачи с помощью Айвики?

Если кратко, то на 4-х ядерном компьютере — нет, а вот на 18-ти ядерном — вполне возможно. Это если не считать того, что бывают задачи, которые не умещаются в память одного компьютера, где просто нет альтернативы распределенной имитации.

На рисунке 3 показан реальный эффект от распараллеливания рассматриваемой задачи с помощью распределенного модуля в сравнении с последовательной версией модели, запущенной уже через последовательный модуль Айвики.

Заключение

В Айвике можно использовать основные парадигмы дискретно-событийного моделирования в рамках одной имитационной модели, включая элементы агентного моделирования. Привычные для многих исследователей методы моделирования, разработанные первоначально для последовательной имитации, Айвика позволяет применять и для распределенной имитации на основе оптимистичного метода деформации времени, где имитация может быть запущена на многоя-

дерном компьютере или кластере компьютеров.

Имитационная модель создается на основе вычислений, которые можно комбинировать, используя вычисления как строительные блоки для создания более сложных вычислений. Это делает запись моделей в Айвике на языке общего назначения Haskell декларативной и близкой к тому, как подобные модели задаются на высокоуровневых специализированных языках, таких как SIMSCRIPT и GPSS.

Функциональное программирование позволяет трактовать эти вычисления как некоторый формализм, с помощью которого можно описывать широкий спектр прикладных задач имитационного моделирования. Причем, формализм обобщается, где последовательная имитация, распределенная имитация и незатронутое в данной работе вложенное моделирование становятся лишь частными случаями единой схемы, реализованной в системе моделирования Айвика.

При всем этом, Айвика — это результат увлечения автором имитационным моделированием на стыке с программированием. По мнению автора, Айвика приобрела законченный вид. В следующих планах автора завершить перенос созданных при работе над Айвикой идей на язык системного программирования Rust, так чтобы достигались следующие цели при неизбежном усложнении программирования моделей:

- сохранились те же моделирующие возможности Айвики;
- повысилась скорость имитации, а также снизилось потребление компьютерных ресурсов, по сравнению с Айвикой;
- систему можно было бы использовать во встраиваемых устройствах;
- имитационные модели можно было бы без дополнительных накладных расходов интегрировать с такими языками программирования, как Python, C, C++, а также с такими математическими пакетами, как Mathematica, Matlab, Maple, Julia, R и т. д.

Если модуль последовательного моделирования практически полностью перенесен на язык Rust и для него цели фактически достигнуты, то для модуля распределенного моделирования, реализующего оптимистичный метод, нужно еще перенести коммуникационную часть, вероятно, с использованием функций MPI для асинхронной передачи сообщений.

Список литературы

1. *Миков А. И., Замятина Е. Б., Фатыхов А. Х.* Система оперирования распределёнными имитационными моделями сетей телекоммуникаций // Труды Первой Всероссийской научной конференции «Методы и средства обработки информации». М.: Изд-во МГУ, 2003. С. 437-443.
2. *Окольнишников В. В.* Представление времени в имитационном моделировании//Вычислительные технологии. 2005. № 10.
3. *Окольнишников В. В.* Разработка системы распределенного имитационного моделирования//Информационные технологии. 2006. № 12. С. 28-31.
4. *Смелянский Р. Л. Бахмуров А. Г., Костенко В. А.* Среда моделирования DYANA: синтез, анализ и оптимизация вычислительных систем реального времени//Сб. докл. I Междунар. конф. «Цифровая обработка сигналов и ее применения». МЦНТИ. 1998. Т. 4. С. 152.
5. *Сорокин Д. Э.* Айвика: имитационное моделирование в терминах вычислений//Труды Седьмой всероссийской научно-практической конференции «Имитационное моделирование. Теория и практика» (ИММОД-2015), 2015.
6. *Сорокин Д. Э.* Устойчивое к разрывам связи распределенное моделирование с Aivika//Труды Восьмой всероссийской научно-практической конференции «Имитационное моделирование. Теория и практика» (ИММОД-2017), 2017.
7. *Chandy M. and Misra J.* Distributed simulation: A case study in design and verification of distributed programs. IEEE Transactions on Software Engineering SE-5, (5): 440-452, 1979.
8. *Cloud Haskell*, URL: http://haskell-distributed.github.io.
9. *Fujimoto R. M.* Parallel and Distributed Simulation Systems. Wiley Interscience, 2000.
10. *Jefferson D. R. and B. Beckman et al.* The Time Warp operating systems//11th Symposium on Operating Systems Principles, 21: 77-93, 1987.
11. OMNeT++ Discrete Event Simulator. URL: https://www.omnetpp.org.