Введение в DVCompute++ Simulator

Сорокин Давид Эрнестович <david.sorokin@gmail.com>,
Россия, Марий Эл, Йошкар-Ола

14 июня 2021 г.

Введение

Дискретно-событийная модель описывает некоторую активность, которая может быть выражена в терминах вычислений [1]. Мы можем определить некоторые простые элементы для представления примитивных вычислений. Затем мы комбинируем эти элементы, чтобы создать вычисления, которые бы зависели от результатов промежуточных вычислений. Это очень общая концепция, которая пришла из мира функционального программирования, но она может быть также применена к дискретно-событийному моделированию.

Более того, таким же способом мы определяем как последовательные, так и распределенные модели. Если вы можете первые запустить на своем ноутбуке, то оптимистичный метод деформации времени (*англ.* Time Warp) и альтернативный консервативный метод позволят вам создавать большой размерности распределенные имитационные модели, которые могут быть запущены на суперкомпьютерах. Везде один и тот же подход работает, одна и та же идея применена.

Как автор этого метода и соответствующей реализации DVCompute++ Simulator, я надеюсь, что вы найдете этот подход не только как теоретическое изобретение, но вы также сможете создавать реальные и полезные модели для практики. Здесь я предполагаю, что читатель знаком с языком программирования C++.

1 Задержка времени

Ключевой момент состоит в том, как мы представляем задержки времени во время имитации. Так, мы можем задержать дискретный процесс на заданный интервал времени, а затем продолжить вычисление (псевдокод):

```
Process<Unit, auto> hold_process(double dt);
```

Здесь функция возвращает некоторый объект типа данных `Process`, который описан сразу же после этого параграфа. Будучи этот объект вовлеченным в имитацию, соответствующий процесс был бы задержан, пока другие моделирующие активности продолжили бы свое выполнение конкурентно. Но чтобы включить этот объект в имитацию, мы должны создать и запустить составное вычисление, которое бы зависело от него.

2 Тип данных процесса

Тип данных `Process` является шаблоном от двух параметрических типов:

```
template<typename Item, typename Impl = internal::process::BoxedImpl<Item>>
class Process;
```

Первый параметрический тип `Item` задает тип данных, значение которого возвращается вычислением `Process`. Например, `Process<int, Impl>` обозначает, что дискретный процесс вычисляет целое число.

Обычно вам не следует беспокоиться о втором параметрическом типе `Impl`. В большинстве случаев этот тип автоматически выводится компилятором C++. Поэтому я использовал ключевое слово `auto` в том псевдокоде, когда описывал выше функцию `hold_process`.

В общем, правило следующее. Если параметрический тип `Impl` обозначает нечто отличное от типа данных `internal::process::BoxedImpl`, которое идет по умолчанию, то это буквально означает, что представляющий дискретный процесс объект, вероятно, размещен на стеке компьютера. Это очень важно для производительности! Однако, если вы видите, что тип данных `Process` параметризован типом `Impl` по умолчанию, то это означает, что данные объекта `Process` находятся в динамической памяти.

Помимо этого, любой объект `Process` может быть преобразован в другой объект `Process`, где второй параметрический тип принимает значение по умолчанию. Другими словами, мы можем перенести данные объекта из стека в динамическую память. Например, это важно для создания рекурсивных вычислений.

```
template<typename Item, typename Impl>
class Process {
  ...
  operator Process<Item>() &&;
};
```

Я буду такие объекты типа данных `Process`, где параметрический тип `Impl` имеет значение по умолчанию, называть *боксовым представлением* объекта, что подразумевает собой, что тип данных `Impl` стерт, а соответствующие данные объекта находятся в куче.

Два амперсанда && означают, что исходный объект *перемещается*, т.е. потребляется, что подразумевает собой, что исходный объект нельзя больше использовать. В коде это может выглядеть примерно так:

```
Process<Item, Impl> comp = ...;
Process<Item> boxed_comp { std::move(comp).operator Process<Item>() };
```

3 Побочные эффекты

DVCompute++ Simulator написан в стиле функционального программирования. Побочные эффекты определены как вычисления, которые, на самом деле, являются шаблонами, где мы должны определить некоторый тип данных для обозначения того, какие значения такое вычисление может вернуть. В большинстве случае побочные эффекты должны вернуть сам факт того, что эффект произведен. Было бы идеальным вернуть экземпляр типа `void`, но C++ запрещает это.

Поэтому определен фиктивный тип данных, чье единственное предназначение параметризовать тип данных `Item` некоторого вычисления таким типом, который бы имел только единственное значение, что можно только создать.

```
struct Unit {
  constexpr Unit() = default;
};
```

Функция задержки `hold_process` возвращает объект типа данных `Process<Unit>`, или более точно, типа данных `Process<Unit, Impl>` для некоторого `Impl`, которое выводится компилятором C++ из определения этой функции. Для краткости, я использовал в псевдокоде тип данных `Process<Unit, auto>`.

4 Создание процессов

Если вы знакомы с концепцией `std::experimental::future` в C++, вы найдете этот материал также знакомым, особенно если вы использовали метод `then`. Такие методы и функции еще называют *комбинаторами*.

Чтобы создать примитивное вычисление `Process`, как говорят, по *чистому* значению, вы можете вызвать одну из следующих функций (псевдокод):

```
template<typename Item>
Process<Item, auto> pure_process(const Item& item);

template<typename Item>
Process<Item, auto> pure_process(Item&& item);
```

Например, вычисление `pure_process(10)` не делает ничего другого, кроме как *вычисления* значения 10, когда оно используется в имитации. Но так сложно понять, почему на самом деле нужно это все, пока мы не введем следующий комбинатор, который имеет долгую историю в мире программирования.

Чтобы связать текущее вычисление с его продолжением, мы можем использовать замыкание:

```
template<typename Item, typename Impl>
class Process {
  ...
  template<typename BindFn>
  inline auto then(BindFn&& k) &&;
}
```

Здесь параметрический тип `BindFn` обозначает замыкание, которое должно принять значение типа данных `Item` и вернуть новое вычисление `Process`. Для представления замыкания шаблонный параметр используется в целях производительности.

Ту же самую идею можно было бы выразить через стандартный тип функций C++, но это было бы сильно менее эффективным! Вот, это уже не настоящее определение:

```
template<typename Item, typename Impl>
class Process {
  ...
```

```
  // N.B. Неэффективно! Настоящий симулятор использует другой подход, как указано выше!
  template<typename ThenItem>
  inline Process<ThenItem> then(std::function<Process<ThenItem>(Item&&)>&& k) &&;
}
```

Так, если мы желаем задержать текущий процесс на 15 единиц модельного времени, а затем еще на 5 единиц модельного времени после первой задержки, то тогда мы можем написать:

```
hold_process(15.0)
  .then([](Unit&& unit) { return hold_process(5.0); })
```

Это уже составное вычисление, которое задерживает текущий процесс на $20 (= 15 + 5)$ единиц модельного времени, когда это используется в имитации. Здесь важно понимать, что вычисление ничего не сделает, пока мы не включим его в имитацию. Позже мы вернемся к этой теме.

Также замыкание должно вернуть вычисление `Process`. Вот, для чего нам нужна упомянутая функция `pure_process`. Если у нас нет готового вычисления, то мы можем создать его.

Для иллюстрации давайте возьмем очень абстрактный пример, где мы берем текущее значение в рамках вычисления `Process` и увеличиваем его на единицу:

```
template<typename Impl>
inline auto inc_process(Process<int, Impl>&& m) {
  return std::move(m)
    .then([](int x) {
      return pure_process(x + 1);
    });
}
```

Тип результата, на самом деле, имеет тип `Process<int, SomeImpl>`, где `SomeImpl` выводится компилятором C++. Если вы помните, объект этого типа данных может быть преобразован к `Process<int>` с типом данных по умолчанию для второго параметрического типа шаблона.

Как обычно, общие концепции трудны для понимания и осознания. Это — действительно общая концепция, известная как *монада*. Например, экспериментальные фьючи в C++ являются монадой. Если вы могли использовать эти вычисления с помощью комбинаторов, чтобы создавать асинхронные программы, то тогда вы сможете создавать и запускать модели с помощью DVCompute++ Simulator. Здесь я просто хочу показать, что DVCompute++ Simulator имеет основательную и мощную теоретическую базу, что означает, что вы сможете моделировать многие и многие имитационные активности.

5 Случайная задержка

В DVCompute++ Simulator существует множество различных функций для порождения случайных значений, но сейчас нам нужна будет одна из них, которая будет использована в примере далее (псевдокод):

```
Process<double, auto> random_exponential_process(double mu);
```

Она задерживает текущий процесс на случайный интервал, распределенный показательно с заданным средним значением. Эта задержка является побочным эффектом вычисления. Однако, само вычисление возвращает значение использованной задержки времени. Это — то, что описывается типом `double` в результирующем вычислении `Process`.

На самом деле, эта функция просто порождает случайное значение для интервала задержки и затем комбинирует его с описанной выше функцией `hold_process`, которая уже реализует задержку. Другими словами, это составное вычисление `Process`. Вскоре мы увидим, как мы можем создавать составные вычисления, чтобы моделировать некоторые полезные активности.

Существует еще другая функция, которая выполняет тот же самый побочный эффект, но которая игнорирует окончательный результат:

```
Process<Unit, auto> random_exponential_process_(double mu);
```

6 Процесс станка

Для иллюстрации введенных выше концепций мы рассмотрим модель станка, который работает в течение некоторого времени наработки, распределенного показательно. Затем станок ломается, и мастер чинит его в течение интервала времени, который тоже распределен показательно. После починки станок продолжает работать. Прошу обратить внимание на то, что это рекурсивно определенное поведение.

```
const double up_time_mean = 1.0;
const double repair_time_mean = 0.5;

Process<Unit> machine_process() {
  return random_exponential_process(up_time_mean)
    .then([](double up_time) {
      return random_exponential_process(repair_time_mean)
        .then([](double repair_time) {
          return machine_process();
        });
    });
}
```

Чтобы соединить разные части друг с другом, мы используем описанный прежде комбинатор `then`. Каждое продолжение запускается только после того, как предшествующая часть вычисления заканчивается. Здесь описано последовательное поведение.

Модельное время меняется скачками, но сам код выглядит линейным. Более того, код определен рекурсивно как бесконечный цикл, но на самом деле он останавливается сразу же после того, как имитация завершается после достижения конечной точки моделирования.

Простой вызов вычисления `machine_process` не запускает дискретного процесса. Как было замечено ранее, результирующее вычисление должно быть еще включено в имитацию, чтобы его запустили. Вскоре мы увидим, как это может быть сделано.

Наконец, в настоящей модели мы могли бы добавить счетчики для сбора статистики по времени наработки и времени починки. Счетчики могут быть реализованы с помощью специальных ссылок, как будет рассказано далее.

7 Обработчики событий

Введенное вычисление `Process` не является единственным в DVCompute++ Simulator. Есть также и другие вычисления.

Представляющее дискретный процесс вычисление `Process` может быть запущенно в рамках вычисления `Event`. Последнее часто используется для представления обработчиков событий. На самом деле, оно подразумевает собой некоторое действие, которое происходит в текущей точке модельного времени. Оно не может быть прервано. Оно не может иметь задержек по времени. Оно не может быть заблокировано и тому подобное. Это — просто функция от времени.

```
template<typename Item, typename Impl = internal::event::BoxedImpl<Item>>
class Event;

template<typename Impl>
Event<Unit, auto> run_process(Process<Unit, Impl>&& comp);
```

Как мы увидим далее, вычисление `Event` может быть использовано для определения моделей в терминах *событийно-ориентированной парадигмы* дискретно-событийного моделирования, тогда как вычисление `Process` является инструментом *процесс-ориентированной парадигмы*.

Каждое моделирующее вычисление имеет соответствующее боксовое представление (т.е. располагающееся в динамической памяти), где параметрический тип `Impl` имеет тип данных по умолчанию. Например, боксовым представлением вычисления `Process` будет просто `Process<Item>` для некоторого `Item`. Похожим образом, боксовым представлением вычисления `Event` является `Event<Item>` для некоторого `Item`. Каждое боксовое вычисление содержит данные, которые расположены в динамической памяти, а параметрический тип `Impl` стерт. Тип `Impl`, отличающийся от типа по умолчанию, обычно указывает на то, что данные соответствующего вычисления находятся на стеке компьютера, но так не всегда.

Ключевой особенностью вычисления `Event` является то, что оно позволяет нам определить обработчик события, который будет запущен в заданной точке времени при условии, что имитация не закончится раньше.

```
template<typename Impl>
Event<Unit, auto> enqueue_event(double event_time, Event<Unit, Impl>&& event);
```

Существуют похожие комбинаторы типа `then` для вычисления `Event`, которые позволяют нам определить большое разнообразие моделирующих активностей. Только задержка по времени должна быть либо представлена с помощью упомянутой функции `enqueue_event`, либо должна быть представлена в терминах вычисления `Process` или более высокоуровневого вычисления `Block`, которое еще не было введено.

Таким образом, класс `Event` является самодостаточным, но каждое вычисление `Process` должно быть запущенно в рамках вычисления `Event`.

8 Запуск имитации

Если мы возьмем некоторое вычисление `Event` и запустим его в начальной точке времени, то тогда мы получим другое вычисление, которое назы-

вается Simulation. Последнее представляет собой некоторое действие, которое происходит в рамках запуска имитации. Оно не связано уже с точкой модельного времени. Оно относится ко всей имитации.

```
template<typename Item, typename Impl = internal::simulation::BoxedImpl<Item>>
class Simulation;

template<typename Item, typename Impl>
class Event {
  ...

  Simulation<Item, auto> run_in_start_time() &&;
  Simulation<Item, auto> run_in_stop_time() &&;
}
```

Если первая функция запускает вычисление в начальном времени, то последняя функция запускает в конечной точке времени имитации.

Это не является наименьшим моделирующим вычислением еще, но это уже то место, где мы, наконец, можем запустить нашу имитацию станка, которую мы написали ранее.

9 Параметры моделирования

Пришло время определить параметры моделирования, по которым мы сможем запустить имитацию. Параметры определяются следующей структурой, которая задает начальное время имитации, конечное время, некоторый небольшой шаг времени и тип генератора случайных чисел, соответственно. Последние два атрибута требуют большего внимания.

```
  Specs specs { 0.0, 1000.0, 0.1, GeneratorSpec() };
```

Пожалуйста, думайте о шаге времени как о временном шаге для интегрирования, как если бы мы использовали метод Эйлера или Рунге-Кутта в случае интегрирования системы дифференциальных уравнений. На самом деле, этот атрибут имеет именно такие корни, но он может быть использован и для других целей, например, для определения размера шага сетки, по которой бы собиралась статистика для графика отклонения (по *правилу трех сигм*). Так, шаг времени следует выбирать достаточно небольшим, но не меньшим. Число шагов должно иметь достаточно разумное значение.

Генератор случайных чисел может возвращать воспроизводимую последовательность чисел для последовательной имитации в случае необходимости, но режим оптимистичной распределенной имитации поддерживает только псевдослучайную последовательность чисел (в некоторых случаях вы можете сами определить воспроизводимый генератор). Последнее задается значением GeneratorSpec(). Оно породит условно невоспроизводимую последовательность псевдослучайных чисел как для режима последовательной имитации, так и для двух поддерживаемых режимов распределенной имитации (оптимистичный и консервативный методы).

По заданным параметрам моделирования мы можем запустить вычисление Simulation, чтобы получить результат.

```
template<typename Item, typename Impl>
class Simulation {
  ...
  Result<Item> run(const Specs* specs) &&;
}
```

Этот метод является именно тем, что запускает всю имитацию!

Таким образом, идея в том, что мы всю имитационную модель представляем как вычисление `Simulation` независимо от того, насколько велика наша модель, а затем вызываем эту простую функцию, чтобы начать имитацию.

Здесь тип `Result` указывает либо на успешный результат, либо на ошибку. Чтобы получить результат или кинуть исключение, мы можем использовать одну из следующих вспомогательных функций:

```
template<typename Item>
const Item& expect_result(const Result<Item>& result);

template<typename Item>
Item& expect_result(Result<Item>& result);

template<typename Item>
Item expect_result(Result<Item>&& result);
```

10 Запуск станков

Ранее мы определили процесс станка. Если мы возьмем два таких станка, то мы можем запустить их в точке начального времени. Итоговая модель будет иметь тип `Simulation<Unit, Impl>` для некоторого `Impl`.

```
auto comp = run_process(machine_process())
  .then([=](Unit&& unit) {
    return run_process(machine_process());
  })
  .run_in_start_time()
  .then([=](Unit&& unit) {
    return pure_event(Unit())
      .run_in_stop_time();
  });
```

Здесь нам действительно нужно запустить пустое вычисление в конечной точке времени так, чтобы два дискретных процесса продолжили бы работать до момента окончания имитации. Иначе бы все активности завершились непосредственно в начальной точке. Однако, запущенное в конечной точке вычисление заставляет очередь событий обрабатывать все обработчики событий. Дискретные процессы также неявно преобразуются в такие обработчики событий.

Последним шагом будет запуск нашей имитации.

```
auto result = std::move(comp).run(&specs);
```

11 Сбор статистики

В DVCompute++ Simulator существуют две структуры для представления сводки о статистической информации. Первая структура имеет название типа данных `SamplingStats`. Она используется для сбора *основанной на наблюдениях статистики* (*англ.* observations-based statistics). Например, это может быть время ожидания в очереди или ресурсе. Вторая структура `TimingStats` уже используется для сбора статистики, которая по-английски называется *time persistent variable statistics*. Например, это может быть статистика по длине очереди в том же ресурсе.

Обе структуры представляют собой неизменяемые типы данных. Обычный подход состоит в том, что мы меняем некоторую ссылку `Ref` в рамках вычисления `Event` для того, чтобы изменить значение статистики. Это довольно-таки эффективно, и это работает как для последовательного режима, так и режимов распределенной имитации.

12 Очередь, многоканальное устройство, прибор и GPSS

В DVCompute++ Simulator есть прибор типа данных `Facility`, который может вытеснить дискретный процесс с меньшим приоритетом транзакта. На самом деле, DVCompute++ Simulator поддерживает так называемый *встроенный предметно-ориентированный язык*, который похож на популярный язык моделирования GPSS, но только модели задаются в терминах языка программирования C++. Это основано на описанном выше вычислении `Process`. Только для представления блоков есть вычисление `Block`, которое может быть запущено в рамках вычисления `Process`. Похожим образом, в DVCompute++ Simulator существует также аналог многоканального устройства из GPSS с типом данных `Storage`, а также аналог очереди `Queue`.

13 Вычисление блока

Вычисление `Block` обозначает некоторую функцию, которая принимает на входе некоторый `Input` и возвращает вычисление `Process<Output>` в рамках некоторого дискретного процесса. В функциональном программировании такая функция называется *стрелкой Клейсли*.

```
template<typename Input, typename Output,
  typename Impl = internal::block::BoxedImpl<Input, Output>>
class Block {
  ...
  template<typename ThenOutput, typename ThenImpl>
  Block<Input, ThenOutput, auto> then(Block<Output, ThenOutput, ThenImpl>&& next) &&;
}
```

Если `a1` и `a2` обозначают некоторые вычисления `Block`, то тогда выражение `std::move(a1).then(std::move(a2))` задает составное вычисление, где мы вначале обрабатываем вход первым вычислением `a1` в рамках дискретного процесса, получаем промежуточный результат, который становится входом для второго вычисления, которое также обрабатывает свой вход в рамках своего дискретного процесса. Следующий результат становится результатом всего составного вычисления.

Эта концепция очень полезна для задания модели способом близким к языку моделирования GPSS. Так, процесс нашего станка мог бы быть также определен как вычисление блока.

```
static Block<SharedRef<Transact<int>>, Unit> machine_chain(const SharedRef<Queue>& queue) {
  return queue_block<int>(queue, 1)
    .then(advance_block<SharedRef<Transact<int>>>(random_exponential_process_(up_time_mean)))
    .then(depart_block<int>(queue, 1))
    .then(advance_block<SharedRef<Transact<int>>>(random_exponential_process_(repair_time_mean)))
```

```
    .then(delay_block([=]() {
      return machine_chain(queue);
    }));
}
```

Здесь тип `SharedRef` похож на стандартный умный указатель языка C++ `std::shared_ptr`, только оптимизированный под выполнение в рамках имитации. Шаблонный класс `Transact<int>` обозначает транзакт, который имеет значение заданного тип данных целых чисел.

Если бы параметры шаблонов могли выводиться компилятором C++, то тогда мы могли бы написать то же самое короче (псевдокод):

```
static BlockChain machine_chain(const SharedRef<Queue>& queue) {
  return queue_block(queue, 1)
    .then(advance_block(random_exponential_process_(up_time_mean)))
    .then(depart_block(queue, 1))
    .then(advance_block(random_exponential_process_(repair_time_mean)))
    .then(delay_block([=]() {
      return machine_chain(queue);
    }));
}
```

Это — бесконечная цепочка блоков, которая моделирует тот же самый процесс станка, где только мы собираем еще статистику, используя экземпляр `queue`.

Наконец, мы должны задержать последнее вычисление так, чтобы мы могли, вообще, создать объект, представляющий вычисление. Мы задерживаем рекурсивный вызов в рамках соответствующего замыкания.

14 Магическая ошибка C1060 для Visual C++

Здесь мы используем так много встраивания кода, что некоторые компиляторы, такие как Microsoft Visual C++, временами протестуют, выдавая магическую ошибку C1060. К счастью, существуют простой обходной путь для этого случая. Мы можем преобразовать вычисление к его боксовому представлению, даже если это снизит скорость кода.

```
static Block<SharedRef<Transact<int>>, Unit> machine_chain(const SharedRef<Queue>& queue) {
  return queue_block<int>(queue, 1)
    .then(advance_block<SharedRef<Transact<int>>>(random_exponential_process_(up_time_mean)))
    .then(depart_block<int>(queue, 1))

#if defined(_MSC_VER)
    // a workaround for the MSVC compiler error C1060
    .operator Block<SharedRef<Transact<int>>, SharedRef<Transact<int>>>()
#endif

    .then(advance_block<SharedRef<Transact<int>>>(random_exponential_process_(repair_time_mean)))
    .then(delay_block([=]() {
      return machine_chain(queue);
    }));
}
```

Это помогает компилятору упростить код, но скорость слегка падает. Поэтому я использовал соответствующее преобразование только для Visual C++ так, чтобы GCC и Clang могли проигнорировать его. На самом деле, Visual C++ должен скомпилировать представленный код, но ошибка действительно возникает в случае слегка более длинных конструкций.

15 Инициализация модели

Наиболее сложной частью с моделирующими вычислениями является то, что они увеличивают уровень вложенности функций с каждым применением комбинатора `then` для монад. Поэтому DVCompute++ Simulator содержит вспомогательный опциональный класс `Model`, который позволяет нам значительно упростить сам процесс инициализации модели.

```
Model model;
```

Мы собираем статистику для станков. Следовательно, нам нужно инициализировать будущий экземпляр `Queue`.

```
auto queue = model.new_queue(std::string("Queue"));
```

Для запуска блоков нам нужны два объекта, которые бы затем были преобразованы в два транзакта для представления станков.

```
auto machine_stream {
  random_int_uniform_stream(0, 0).take(2)
};
```

Здесь нули означают, что поток порождает бесконечное количество данных в начальной точке, но мы берем только два первых значения. Так, итоговый поток имеет размер два.

Теперь, чтобы спланировать запуск станков, мы можем создать транзакты по заданному потоку двух элементов и затем обработать оба транзакта введенной ранее цепочкой блоков.

```
auto machine_start {
  run_process(stream_generator_block<int>(std::move(machine_stream), 1)
      .run([=]() { return machine_chain(queue); }))
    .run_in_start_time()
};
```

Здесь транзакты имеют приоритет 1 для примера. Реальная величина приоритета не имеет значения для этой конкретной модели.

Мы только создали вычисление `Simulation<Unit>`. Теперь мы добавляем его ко вспомогательному объекту модели для того, чтобы имитационная модель запустила станки во время своей инициализации.

```
model.emplace_action(std::move(machine_start));
```

Наконец, мы инициализируем модель в рамках `Simulation<Unit>` и возвращаем некоторую статистику в конечной точке времени, например, среднее значение содержимого очереди.

```
auto comp {
  std::move(model)
    .init_in_start_time()
    .then([=](Unit&& unit) {
      return queue_content_stats(queue)
        .map([](SamplingStats<int>&& x) { return x.mean; })
        .run_in_stop_time();
    })
};
```

Сейчас мы запускаем всю имитацию, чтобы получить соответствующую статистику для двух заданных станков.

```
auto result = std::move(comp).run(&specs);
auto x = expect_result(result);
```

16 Сигналы

Внутри самого DVCompute++ Simulator широко используется шаблонный класс `Observable` для реализации сигналов, но вы также можете использовать его в своих моделях. Вычисления `Event` и `Observable` могут быть использованы для моделирования аппаратуры, такой как цифровые интегральных микросхемы. Обработчики событий могут иметь приоритеты, которые могут быть использованы для синхронизации сигналов. Сигналы также могут быть задержаны через очередь событий, соединены или разъединены.

17 Оптимистичное распределенное моделирование

Все описанное выше применимо и для последовательного моделирования, и для распределенного моделирования на основе оптимистичного метода деформации времени (*англ.* Time Warp). Вы можете запустить имитацию на своем ноутбуке, но вы также можете создать распределенную модель и затем запустить ее на суперкомпьютере с помощью DVCompute++ Simulator. В последнем случае используется протокол MPI для эффективного взаимодействия между *логическими процессами*.

Разница между обоими режимами в том, что вы должны использовать разные пространства имен C++, но моделирующие вычисления и сущности имеют те же названия. Только в случае распределенной имитации эти вычисления и сущности часто требуют, чтобы типы и замыкания были копируемыми, т.е. они должны иметь конструктор копирования, тогда как последовательный режим имитации почти целиком опирается на семантику перемещения.

Требуется операция копирования, поскольку распределенная имитация может иметь откаты. Поэтому вычисления не могут быть необращаемыми. Каждый раз, как очередь событий должна запустить обработчик события, она копирует этот обработчик для создания его клона. Обычно это довольно-таки дешевая операция. Возможно, необходимость клонировать является наиболее значительным отличием последовательного режима и режима оптимистичной распределенной имитации в DVCompute++ Simulator. В других аспектах оба режима очень похожи.

Помимо этого, в случае распределенной имитации вы создаете логические процессы, которые могут посылать сообщения друг другу. Действие отсылки сообщения выражается в терминах вычисления `Event`, тогда как действие получения сообщения естественным образом выражается в рамках вычисления `Observable`.

18 Консервативное распределенное моделирование

DVCompute++ Simulator поддерживает также консервативный метод распределенного моделирования. Здесь те же самые функции и те же самые

моделирующие концепции работают, как и прежде! Только мы должны задать так называемый параметр *предсказания* (*англ.* lookahead), который является границей снизу для минимальной возможной задержки в единицах модельного времени, что может возникнуть при передаче сообщений между логическими процессами.

Важны оба метода распределенного моделирования. Оба метода поддерживаются в DVCompute++ Simulator.

19 Вложенное моделирование

Наконец, есть другой режим моделирования, который поддерживается в DVCompute++ Simulator, где все введенные прежде моделирующие концепции остаются в силе. Это — одна из версий вложенного моделирования, где мы можем создать подмодели в рамках любой модели или подмодели в любой точке модельного времени, вызвать эти подмодели в их будущем модельном времени, получить некоторую оценку, а затем вернуться к текущей модели или подмодели для продолжения выполнения с текущей точки модельного времени, используя полученную оценку. Такие вложенные модели обычно имеют экспоненциальную сложность. Поэтому уровень ветвления таких подмоделей должен быть ограничен некоторым значением.

Этот режим имитации связан с теорией игр. Он может быть использован для нахождения некоторой оптимальной стратегии поведения. На взгляд автора это крайне интересная область, которая еще не вполне изучена.

20 Единый метод моделирования

Тем не менее, все рассмотренные режимы основаны на том же самом методе моделирования. После того, как вы попрактикуетесь с простейшим режимом среди них, т.е. с режимом последовательной имитации, вы сможете начать создавать распределенные или вложенные имитационные модели. Это может быть увлекательным!

21 Лицензия

DVCompute++ Simulator бесплатен для некоммерческого использования, но для использования симулятора в коммерческих проектах вы должны будете приобрести коммерческую лицензию, связавшись со мной по почте david.sorokin@gmail.com.

Список литературы

[1] Сорокин Давид Эрнестович. Распределенное имитационное моделирования с Aivika. *Прикладная информатика*, №4 (82), 2019.