Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования  
Пермский государственный национальный исследовательский университет

Кафедра математического обеспечения вычислительных систем

Алгоритмы синхронизации агентов в системе моделирования «Better»

Выпускная квалификационная работа магистра

Работу выполнил студент группы «ПМИ‑1,2‑2011НМ» 2-го курса магистратуры механико-математического факультета  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Митраков А.А.

Научный руководитель:  
Доцент, кандидат физико-математических наук  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Замятина Е.Б.

“\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Пермь, 2013

Аннотация

Данная работа посвящена алгоритмам синхронизации логических процессов для систем распределённого агентного моделирования. В области имитационного моделирования существуют два обширных класса алгоритмов: консервативные и оптимистические, причём в основном выбор делается в пользу последних. Однако оптимистические алгоритмы не лишены недостатков, которые специалисты и эксперты и по сей день пытаются нивелировать.

В работе предложен новейший подход к оптимизации данного класса алгоритмов, основанный на знаниях об исследуемой модели. В тексте представлены описания существующих и предлагаемых алгоритмов, оценка сложности, область их применения, практические результаты. Также описана архитектура агентной платформы «Better», на которой проводились эксперименты.

Оглавление

[Введение 4](#_Toc357947979)

[Глава 1. Преимущества агентного имитационного моделирования 6](#_Toc357947980)

[Имитационное моделирование 6](#_Toc357947981)

[Мультиагентные системы 8](#_Toc357947982)

[Агентное моделирование 10](#_Toc357947983)

[Глава 2. Постановка задачи. Проблема синхронизации логических процессов 12](#_Toc357947984)

[Распределённое имитационное моделирование 12](#_Toc357947985)

[Консервативные алгоритмы синхронизации 14](#_Toc357947986)

[Оптимистические алгоритмы синхронизации 17](#_Toc357947987)

[Сравнение классов алгоритмов 19](#_Toc357947988)

[Глава 3. Оптимизация алгоритмов синхронизации логических процессов 22](#_Toc357947989)

[Модификации консервативных алгоритмов 22](#_Toc357947990)

[Модификации оптимистических алгоритмов 23](#_Toc357947991)

[Методы, направленные на устранение побочных эффектов оптимистических алгоритмов 25](#_Toc357947992)

[Глава 4. Алгоритмы синхронизации агентов, основанные на знаниях о модели 26](#_Toc357947993)

[Ограничения и сфера применимости алгоритмов 26](#_Toc357947994)

[Природа откатов 28](#_Toc357947995)

[Алгоритм#1 29](#_Toc357947996)

[Алгоритм#2 30](#_Toc357947997)

[Алгоритм#3 33](#_Toc357947998)

[Глава 5. Разработка платформы. Архитектура, технологии и инструментальные средства 47](#_Toc357947999)

[Модель акторов 47](#_Toc357948000)

[Глава 6. Способы представления знаний о модели. Разработка DSL 50](#_Toc357948001)

[Глава 7. Эксперименты. Практические результаты 51](#_Toc357948002)

[Заключение 52](#_Toc357948003)

[Библиографический список 53](#_Toc357948004)

[Глоссарий 54](#_Toc357948005)

[Приложение 55](#_Toc357948006)

Введение

Данная работа посвящена разработке системы распределённого агентного моделирования и исследованию алгоритмов синхронизации агентов. Агентное моделирование является сравнительно новым и весьма перспективным направлением в области имитационного моделирования, поскольку данный подход способен решать очень сложные и трудноформализуемые задачи, такие как моделирование социальных, экономических и политических процессов, сложные динамические и стохастические задачи в эпидемиологии, экологии, производстве, логистике, транспорте, военно-стратегическом планировании и т.д.

Основными предпосылками к развитию агентного моделирования являются развитие методов инженерии знаний и искусственного интеллекта, а также совершенствование мощностей аппаратного обеспечения. Первое послужило причиной перехода от классических реактивных агентов к интеллектуальным – способным функционировать на основе заложенных в него знаний, принимать решения в нестандартных ситуациях и самообучаться в динамике. Это позволило моделировать такие сложные сущности, как человек, фирма, социальная группа и государство.

Развитие аппаратуры играет не менее важную роль – в силу своей интеллектуальности, агенты могут занимать большие объёмы вычислений. Для традиционных методов моделирования, например, системной динамики, не имеет значения масштаб моделируемого объекта – покупатели в супермаркете или совокупность жителей мегалополиса. Для агентного подхода, описывающего поведение «снизу вверх», это является кричным показателем.

В настоящее время сложные вычислительные задачи решаются с помощью распределённых систем и специального класса алгоритмов. Агентное моделирование хорошо подходит для распараллеливания (как на уровне данных, так и на уровне задач), поэтому распределённость – одно из главных требований к платформе. За основу модели имитации взята идея распределённого дискретно-событийного моделирования – множество логических процессов продвигают время, обрабатывают события и обмениваются друг с другом сообщениями.

Одним из ключевых алгоритмов для поддержки распределённого дискретно-событийного моделирования является алгоритм синхронизации времени. Проблема заключается в том, что увеличение количества вычислительных узлов в N раз в идеале должна привести к соответствующему приросту в скорости вычислений в N раз, однако на практике это далеко не так. Приходится платить за коммуникацию агентов, неравномерность производительности и, разумеется, алгоритмы синхронизации.

Синхронизации времени являются самым узким местом в распределённом моделировании. Существует немало исследований, преимущественно зарубежных, о разработке новых методов синхронизации и совершенствованию уже существующих. Однако до сих пор *не найдено идеального решения*, полностью устраняющего ожидания в консервативных алгоритмах, либо устраняющего откаты в оптимистических. Данные алгоритмы порождают и другие проблемы – нехватку памяти, переполнение сети служебными сообщениями и т.д.

Целью работы является разработка агентной платформы и реализация нового оптимистического алгоритма синхронизации агентов на базе данной платформы. Практика показала, что наиболее прогрессивными являются те алгоритмы, которые используют знания о модели (lookahead, lookback, знания протокола сообщений, оценка следующего события). Предложенный алгоритм не только использует явно специфицированные знания о модели, но и выводит новые знания в ходе имитационного прогона.

Теоретическая значимость работы заключается в повышении эффективности параллелизма работы распределённых агентных платформ за счёт применения специальных алгоритмов синхронизации. Практическая ценность работы заключается в сокращении накладных расходов при синхронизации агентов, что приводит к снижению затрат по памяти, по процессорному времени и по нагруженности коммуникационной среды.

1. Преимущества агентного имитационного моделирования

В данной главе будет представлен краткий экскурс в область агентного моделирования: предпосылки возникновения, актуальность, сфера применения. Также будут показаны общие и отличительные признаки агентного моделирования по сравнению с традиционными методами машинной имитации.

Имитационное моделирование

Моделирование – один из способов познания окружающего мира, при использовании которого исследуемая система заменяется более простым объектом, описывающим реальную систему с заданной степенью точности.

Моделирование применяется в случаях, когда проведение экспериментов над реальной системой невозможно, затруднительно, дорого, нецелесообразно или сопровождается риском для аппаратуры или жизни и здоровья человека. Также моделирование возможно использовать для ускорения получения статистических данных о состоянии реального объекта за длительный период времени.

Различают физическое и математическое моделирование. Физической моделью может служить уменьшенная копия исследуемого объекта (например, модель башенного крана). При использовании математического моделирования поведение системы описывается с помощью формул. Из математического моделирования отделилась новая форма исследования – имитационное моделирование.

Имитационная модель – это программа, которая описывает структуру и воспроизводит поведение реальной системы во времени. В основном, имитационное моделирование применяют для исследования особым образом организованных сложных динамических систем с элементами случайности и множеством входных параметров за некоторый промежуток времени. В ходе моделирования собирается статистика о различных аспектах функционирования системы. Затем процесс имитации анализируется и повторяется для выявления зависимостей от входных параметров.

Зачастую наряду с понятием «имитационное моделирование» применяют термин «машинная имитация».

Область применимости имитационного моделирования

Ярким примером задачи, решаемой методами машинной имитации, является моделирование систем массового обслуживания. Хотя задача и может показаться специализированной, схожие проблемы возникают во многих областях науки, техники, производства, логистики, культуры, туризма и сервиса.

Оплата времени квалифицированного работника и времени использования аппаратуры составляет немалую долю расходов компаний. Определение оптимального графика использования ресурсов, позволяющего системе эффективно выполнять поставленные задачи, позволяет снизить расходы, учесть риски и повысить надёжность реальных систем.

Но также имитационное моделирование может применяться и для других задач: исследование биологических, социальных, экономических явлений, прогнозирование поведения сложных динамических систем, моделирование техногенных аварий и катастроф, выявление закономерностей и тенденций в различных временных процессах.

Преимущества имитационного моделирования

Применение имитационных моделей дает множество преимуществ по сравнению с выполнением экспериментов над реальной системой и использованием других методов.

*Стоимость*. Зачастую построить имитационную модель гораздо дешевле, чем нанимать персонал, закупать реальное оборудование, привлекать службы экспертов и специалистов. Модель можно сколь угодно много раз прогнать заново, изменяя и регулируя входные параметры.

*Время*. В реальности оценить эффективность нововведений можно лишь через месяцы или даже годы. Имитационная модель позволяет определить оптимальность таких изменений за считанные минуты, необходимые для проведения эксперимента.

*Воспроизводимость*. Современная жизнь требует от организаций быстрой реакции на изменение ситуации на рынке. Любые малейшие изменения потребуют и быстрого изменения в структуре производства. С помощью имитационной модели можно провести неограниченное количество экспериментов с разными параметрами, чтобы получить наилучший результат.

*Точность*. Традиционные расчетные математические методы требуют применения высокой степени абстракции и не учитывают мелкие детали. Имитационное моделирование позволяет описать структуру системы и её процессы в естественном виде, не прибегая к использованию формул и строгих математических зависимостей.

*Наглядность*. Имитационная модель обладает возможностями визуализации процесса работы системы во времени, схематичного задания её структуры и выдачи результатов в графическом виде. Это позволяет наглядно представить полученное решение и донести заложенные в него идеи до клиента или заказчика.

*Универсальность*. Имитационное моделирование позволяет решать задачи из широкого круга областей: от исследования покупателей в супермаркете до имитации сложных экономических и политических процессов, где стандартные приёмы исследования далеко не всегда дают адекватный результат.

Мультиагентные системы

С развитием информационных технологий, ростом информационных ресурсов, совершенствованием сетевой инфраструктуры и увеличением производительности вычислительных машин появляются всё новые и новые задачи. Более сложные, более ресурсозатратные и менее формализуемые. Новые задачи требуют поиска новых подходов и методов решения. Порою задачи бывают такими, что использование традиционных подходов и алгоритмов неприменимо в силу специфики области применения: недетерминированности, постоянной изменчивости, слабоформализуемости и т.д.

Возникла необходимость в более мощных и гибких интеллектуальных программных системах, способных непрерывно приобретать новые знания и изменять свою структуру и функции, развиваясь и адаптируясь к решаемым задачам и условиям внешней среды.

Одним из путей решения подобных задач можно назвать применение мультиагентных систем (МАС) – особой формы области искусственного интеллекта, которая базируется на знаниях и эвристических алгоритмах кооперативного поиска решения задачи.

Ключевым элементом этих систем становится программный агент, способный воспринимать ситуацию, принимать решения и взаимодействовать с другими агентами. В основу успешного коллективного решения задачи положено 3 фундаментальных принципа: кооперация, координация и коммуникация агентов.

В упрощённом виде можно представить агента как некоторую сущность, обладающую памятью и собственной базой знаний, умеющей находить решение некоторой узкой специфической задачи, взаимодействовать с другими агентами и менять правила поведения в динамике. Следует особо оговорить, что ни один агент в МАС не способен решить общую задачу самостоятельно.

Агента можно создать таким образом, что он будет иметь определенное отношение к принятию рискованных решений в условиях неопределенности. Агенты функционируют в едином виртуальном мире. В ходе переговоров агентов формируется текущее решение проблемы, которое гибко меняется в соответствии с динамикой среды.

Существует множество определений агента. Но обычно агентов определяют через свойства, которыми те должны обладать, а именно:

* автономность – способность действовать без внешнего управляющего воздействия и осуществлять контроль собственных действий и внутреннего состояния;
* активность – способность ставить цели и выполнять заданные действия для их достижения;
* реактивность – адекватное восприятие состояния среды и реакция на его изменение;
* коммуникативность – взаимодействие с другими агентами;
* целенаправленность – предполагает наличие собственных источников мотивации;
* открытость – любой агент представляет собой открытую систему;
* базовые знания – знания агента о себе, других агентах, окружающей среде.

Для достижения целей интеллектуальные агенты взаимодействуют друг с другом, устанавливают связь между собой и выполняют заданные действия или операции в соответствии с имеющимися целями и намерениями. Более того, агент, выполняя заданные действия, изменяет состояние среды и учитывает возможность возникновения нестандартных состояний (тупиков, отсутствие ресурса и др.).

В общем случае среда, где функционирует агент, имеет определенное состояние и поведение, которые могут быть известны полностью или частично. Агент познаёт среду, приспосабливается к ней и изменяет её своими действиями.

В отличие от традиционных систем, в которых решение ищется с помощью централизованных, последовательных и детерминированных алгоритмов, в мультиагентных системах решение достигается в результате распределённого взаимодействия множества агентов, ориентированных на поиск не столько оптимального, сколько наилучшего из возможных решений на текущий момент времени.

Эти новые принципы радикально отличают МАС от классических алгоритмов и позволяют применять агентов для решения крайне сложных задач.

Агентное моделирование

Классическое имитационное моделирование, объединяясь с мультиагентными системами, породило принципиально новую ветвь научного исследования – агентное моделирование.

В литературе можно найти множество различных определений агентного моделирования. В целом агентное моделирование можно определить как метод имитационного моделирования, исследующий поведение децентрализованных агентов и то, как это поведение определяет поведение всей системы в целом. При разработке агентной модели, инженер вводит параметры агентов, определяет их поведение, помещает их в некую окружающую среду, устанавливает возможные связи, после чего запускает моделирование. Индивидуальное поведение каждого агента воплощает глобальное поведение моделируемой системы.

Традиционные подходы имитационного моделирования рассматривают объекты как некоторые пассивные заявки (транзакты) в процессе. Например, модели системной динамики полны предположений о глобальных законах распределения, пуассоновских потоках клиентов и т.д. В процессном моделировании (дискретно-событийное моделирование) имитация рассматривается как совокупность процессов, объединяющих набор дискретных событий. Эти методы превосходят “аналитическое моделирование” в возможности учитывать случайность, динамику и нелинейность, но они проигрывают в том факте, что сами объекты и их взаимосвязи могут быть абсолютно различными.

В действительности, в классике поток клиентов рассматривают как некоторый марковский процесс. Однако люди могут быть с различными доходами и интересами, они могут иметь разную производительность труда; могут взаимодействовать и конкурировать, могут зависеть один от другого; могут консультироваться с друзьями или прохожими, могут принимать ошибочные решения.

Итак, мы переходим к агентному моделированию – абсолютно новому подходу в исследовании сложных динамических систем, интегрирующему в себе методы имитационного моделирования и мультиагентных систем.

Агентное моделирование избавлено от описанных выше ограничений, поскольку оно предполагает сосредоточение непосредственно на отдельных объектах, их поведении и коммуникации. Агентная модель – это ряд взаимодействующих активных объектов, которые отражают объекты и отношения в реальном мире. Таким образом, агентное моделирование делает шаг вперед в понимании и управлении совокупностью сложных социальных и бизнес процессов.

Фундаментальное отличие агентного моделирования состоит в том, что модель строится не «сверху вниз», как в системной динамике или процессном моделировании, а «снизу вверх». Таким образом, исследователю не требуется знать глобальных законов распределения. Вместо этого сами агенты за счёт своего персонального поведения формируют целостную картину об исследуемом процессе или явлении.

Область применения агентного моделирования

Хороший пример использования агентного моделирования – потребительский рынок. В очень динамичной, конкурентной и сложной среде рынка выбор покупателя зачастую зависит от индивидуальных особенностей, врожденной активности потребителя, сети контактов, а также внешних влияний, которые лучше всего описываются с помощью агентного моделирования.

Другой пример – моделирование мирового экономического кризиса, связанного с политикой ведения хозяйства различных стран. Все государства преследуют свои собственные цели, они ведут переговоры, налаживают торговые связи, вступают в конфликты, заключают торговые сделки. Вряд ли политику государства, чувствительную к любым изменениям как в мире, так и внутри страны, можно описать с помощью математических формул.

Однако не следует думать, что агентное моделирование применимо только для решения задач коммуникативного характера. Задачи, связанные с логистикой, производством, цепочками поставок или бизнес-процессами, также решаются с помощью агентного моделирования. Например, организация производства на предприятии с управлением ресурсами, движением сырья и планированием бюджета может быть эффективно исследована с помощью агентов.

Ещё одна отрасль – исследование социальных процессов. Например, моделирование распространения инфекции в случае эпидемий, выявление автомобильных заторов на городских улицах, исследование эвакуации людей с места техногенной аварии могут оказаться под силу лишь агентному подходу к моделированию.

1. Постановка задачи. Проблема синхронизации логических процессов

В данной главе мы рассмотрим переход от последовательного моделирования к распределённому, выявим проблемы, возникающие в процессе данного перехода и покажем, какие существуют алгоритмы (классы алгоритмов) для их решения. В конце главы будет показано, какими недостатками обладают эти алгоритмы, и как их можно было бы преодолеть.

Распределённое имитационное моделирование

Первоначально имитационное моделирование, зародившись в 50-х гг. прошлого столетия, представляло собой последовательный симулятор (все вычисления проводились на одном компьютере). Появлялись новые формы функционирования имитационного моделирования (системная динамика, непрерывное моделирование, дискретно-событийное моделирование), но модель вычислений оставалась прежней – последовательной.

Рассмотрим подробнее схему последовательного симулятора на примере дискретно-событийного моделирования (*SDES*, от англ. Sequential Discrete Event Simulation).

[рис.]

В системе имеется совокупность объектов моделирования. Это могут быть потоки заявок (транзактов) для процессо-ориентированных систем, агрегаты и вентили для системной динамики, агенты для агентного моделирования и т.д. Отдельная компонента – механизм продвижения времени – выполняет планировку (диспетчеризацию) событий: у отдельных объектов просматриваются локальные списки событий (календарь событий) и выбирается событие с наименьшей временнóй меткой. Далее это событие обрабатывается, возможно, порождая новые события, и далее этот алгоритм зацикливается.

[рис.]

Остановом моделирования могут служить либо пустые списки событий (все события обработаны), либо специальные условия (окончание времени моделирования, наступление «особого» события и т.п.).

Какие можно предложить оптимизации и усовершенствования данных симуляторов? Первая и, пожалуй, единственная практически значимая модификация – совершенствование структур данных, применяемых в списках событий. Очевидно, соответствующие структуры данных должны учитывать 2 важных свойства:

* события должны быть упорядочены по времени (или, по крайней мере, должен быть реализован эффективный поиск минимума, как, например, в Binary-Search-Tree);
* должна быть реализована эффективная вставка в конец (поскольку новые события будут иметь временнýю метку больше, чем обработанные).

На рис. приведён пример структуры данных для списка событий (хорошее, но не самое эффективное решение)

[рис.]

На этом, видимо, перечень модификаций заканчивается. Следующим шагом послужило появление распределённого моделирования.

Специфика распределённого моделирования

Параллельное имитационное моделирования зародилось в конце 70-х гг прошлого столетия. Впредь мы будем рассматривать распределённое имитационное моделирование на примере дискретно-событийного моделирования (PDES, от англ. Parallel Discrete Event Simulation).

Общая идея достаточно проста и очевидна: теперь моделируемые сущности выполняются не на одном вычислительном узле, а на 2-х и более. Это могут быть системы с общей памятью (SMP-архитектуры), машины с массовым параллелизмом (MPP-архитектуры), несколько узлов в составе кластера, NUMA-платформы или суперкомпьютеры.

Каждый моделируемый объект также сохраняет локальный календарь событий. Схема функционирования не изменяется – выбирается событие с наименьшей временнóй меткой, далее оно обрабатывается, порождая новые события. На практике число имеющихся вычислительных узлов бывает много меньше числа моделируемых сущностей, в связи с чем последние объединяются внутри одного логического процесса.

Определение. *Логический процесс* – независимая обособленная функциональная единица распределённого дискретно-событийного моделирования, выполняющаяся на 1-м вычислительном узле и связанная с др. процессами в единую сеть посредством передачи сообщений.

К настоящему времени специалисты выделяют два подхода к организации параллельных средств машинной имитации:

* монолитные системы, в которых моделирование сводится к взаимодействию совокупности логических процессов;
* программные компоненты, которые позволяют объединить существующие средства имитации в одну общую сеть для реализации параллельного моделирования.

К последним относят технологию HLA (High Level Architecture), которая де-факто объявлена стандартом распределённого моделирования в США. К агентным системам, поддерживающим HLA, относят “HLA-AGENT” и “SIMAGENT TOOLKIT”.

Проблема локальной каузальности

Основной фундаментальной проблемой распределённого моделирования, к которой данная работа имеет непосредственное отношение, является нарушение локальной каузальности.

Определение. Под *локальной каузальностью* понимается гарантия того, что все события в ходе имитационного прогона будут выполнены по возрастанию (или, строго говоря, неубыванию) временных меток.

Иными словами, обеспечение локальной каузальности гарантирует, что все события будут обработаны в хронологическом порядке. В последовательных машинах таковой проблемы не было: симулятор всегда выбирал минимальное событие. В параллельных системах все логические процессы независимы, а значит, возможна ситуация обработки события «из прошлого».

Пример нарушения локальной каузальности проиллюстрирован на рис.

[рис.]

Если локальные часы логического процесса имеют время t1, а в это время поступает сообщение с временнóй меткой t2, то при t2≤t1, очевидно, нарушается локальная каузальность (поступает событие «из прошлого»).

На помощь нам приходят 2 обширных класса алгоритмов синхронизации: консервативные и оптимистические.

Консервативные алгоритмы синхронизации

Консервативные алгоритмы появились исторически раньше своих оптимистических аналогов. Впервые подобный алгоритм был описан специалистами Chandy, Misra и Bryant в 1977-1979 гг.

Общая идея данного семейства алгоритмов – не позволять логическому процессу продвигать время вперёд, пока он не убедится, что все остальные процессы достигли этого времени. Другими словами, процесс блокируется до тех пор, пока он точно не будет знать о том, что выполнение следующего события безопасно.

Пример работы консервативного алгоритма приведён на рис.



Рис. . Схема работы консервативного алгоритма

Из рисунка видно, что логический процесс не будет выполнять событие с t=5, поскольку теоретически возможна такая ситуация, что LP3 пришлёт событие с меткой t=4. Как только процесс получит сообщение с меткой t≥5, он увеличивает LBTS, а значит, сможет продолжить работу.

Определение. *LBTS* (от англ. «Lower Bound of TimeStamps» – нижняя граница временных меток) – время, равное минимуму временных меток по всем входным очередям сообщений.

Формальная цель данного класса алгоритмов – исключить теоретическую возможность нарушения локальной каузальности.

Проблемы, связанные с реализацией консервативных алгоритмов

Основной проблемой консервативных алгоритмов является теоретическая возможность взаимных блокировок (deadlocks). В связи с этим, консервативный алгоритм без разрешения данного конфликта вообще не может считаться алгоритмом (в силу определения последнего).

Пример возникновения тупика изображён на рис.

[рис.]

Простейший способ разрешения тупиковых ситуаций был предложен авторами базового алгоритма []. Подход называется «Консервативный алгоритм с нулевыми сообщениями». В области распределённого имитационного моделирования данный метод является классикой.

Основные ограничения алгоритма:

* Топология сети фиксирована и известна каждому логическому процессу
* Каждый логический процесс высылает сообщения с неубывающими временными метками
* Коммуникационная среда гарантирует, что все сообщения придут строго в порядке их отправления

Суть данного алгоритма заключается в том, что логические процессы после каждой обработки события высылают всем соседям сообщение специального типа – нулевые сообщения (Null Messages). Их временнáя метка t=a0 свидетельствует о том, что процесс гарантированно в будущем не пришлёт сообщений с меткой t<a0.

Определение. *Нулевое сообщение* – сообщение, несущее в себе только информацию о временнóю метке.

При получении нулевого сообщения логический процесс увеличивает LBTS, что делает все локальные события с t≤LBTS безопасными. Далее он обрабатывает безопасное событие (ели таковое имеется) и вновь высылает всем соседям нулевые сообщения.

Пример работы алгоритма изображён на рис.



Рис. . Консервативный алгоритм с нулевыми сообщениями

Алгоритм оставляет открытым один вопрос: как оценить временнýю метку нулевого сообщения? Одно из возможных решений – применение lookahead («забегание вперёд») []. В этом случае после обработки события с меткой t=t0 логический процесс отправит нулевые сообщения с меткой t=t0+lookahead. В качестве значения lookahead могут использоваться оценки физических задержек устройств, знание нижнего предела в потоке заявок, дискретизация времени и т.д.

Возможные пути оптимизации консервативных алгоритмов будут сформулированы в главе 3.

Оптимистические алгоритмы синхронизации

Оптимистические алгоритмы являются вторым классом алгоритмов синхронизации. Первый алгоритм под названием «Time Warp» предложил Jefferson [] в 1985г.

Оптимистические алгоритмы, в отличие от консервативных, не блокируют управление логического процесса, а просто продвигают время вперёд. В случае нарушения локальной каузальности специальные механизмы выполняют откат текущего состояния модели на прежнее согласованное состояние.

Так, например, в случае отката, вызванным событием с меткой t=t0, алгоритм должен вернуть состояние модели ***x*** на время:

где *Х* – множество состояний модели

*t(x)* – временнáя метка состояния *х*

Иными словами, выбирается состояние, предшествующее по времени сообщению с меткой t0.

Сообщение «из прошлого» в зарубежной литературе называется «Straggler Message» (в отечественной литературе термин официально не утверждён). Общая схема работы алгоритма изображена на рис.

[рис.]

В отличие от консервативного алгоритма с нулевыми сообщениями, Time Warp не накладывает ограничений ни на структуру и топологию сети, ни на качество коммуникационной среды (доставка должна быть гарантирована, но порядок получения сообщений из канала не обязан совпадать с порядком отправления).

Проблемы, связанные с реализацией оптимистических алгоритмов

Одной из проблем такого алгоритма является необходимость восстановления любого предыдущего состояния модели. Основными методами решения могут служить следующие:

* хранение моментальных снимков модели;
* обратные вычисления.

Очевидно, в первом случае алгоритм приводит к значительным издержкам по памяти, а во втором – по процессорному времени. Более того, обратные вычисления, в свою очередь, обладают ещё двумя недостатками:

* требуются наличие специальных алгоритмов и процедур, способных для каждого f вычислить f-1;
* существуют такие функции f, что вычисление f-1 трудозатратно или невозможно (так называемые «односторонние» функции).

Другой проблемой алгоритма являются сообщения, отправленные непосредственно перед процедурой отката. Пример продемонстрирован на рис.

[рис.]

Действительно, если логический процесс отправил сообщение с временнóй меткой t=10 и обработал событие с t=12, а затем получил сообщение «из прошлого» с меткой t=8, то он обязан откатиться назад на время t≤8. Что делать с отправленным сообщением? Jefferson в своей работе предложил идею применения антисообщений (Antimessages).

Определение. *Антисообщение* – сообщение, отменяющее ранее высланное сообщение.

Итак, если процесс получает антисообщение, то возникает одна из следующих ситуаций:

1. Если временнáя метка антисообщения больше текущего времени процесса, то можно заключить, что «позитивное» сообщение ещё находится во входящей очереди. В этом случае оба сообщения взаимно уничтожаются (а сам симулятор даже не узнает об их существовании).
2. Если временнáя метка антисообщения меньше или равна текущему времени процесса, то, согласно общему правилу, возникает откат.

Следует учесть, что в последнем случае откат, вызванный антисообщением, может породить новые антисообщения, которые могут вызвать откаты на других процессах, которые могут привести к новым антисообщениям и т.д.

Определение. *Каскадный откат* – откат, порождённый принятым от другого логического процесса антисообщением. Количество таких цепочек откатов называют *порядком каскадного отката*.

Сравнение классов алгоритмов

Каковы же критерии выбора того или иного алгоритма? Как правило, выбор зависит от конкретной задачи, но в целом оптимистические алгоритмы показали бóльшую эффективность (под эффективностью будем понимать время вычисления имитационного прогона).

Консервативные алгоритмы тратят много времени на ожидание – по факту, система представляют собой «псевдопараллельный» симулятор. Тем не менее, на практике таковые до сих пор применяются. В основном, это связано с многочисленными недостатками оптимистических алгоритмов, которые в силу тех или иных причин оказываются критичными в конкретных системах.

С другой стороны, если исследуемая модель располагает весьма эффективными оценками lookback, то, возможно, консервативный алгоритм будет наилучшим решением. Также отметим, что в плане реализации консервативные алгоритмы зачастую проще в разработке, отладке и сопровождении.

Прежде чем переходить к сравнительной характеристике, рассмотрим ещё несколько недостатков алгоритмов синхронизации, которые являются следствием уже перечисленных недостатков.

Побочные эффекты от применения алгоритмов синхронизации

В предыдущих параграфах были отражены лишь общие проблемы консервативных и оптимистических алгоритмов. В данном разделе рассмотрим ещё некоторые неочевидные проблемы.

Во-первых, в консервативных алгоритмах возможна перегрузка сети нулевыми сообщениями. В классическом варианте логический процесс должен после каждого обработанного события вычислить оценку LBTS и отправить нулевые сообщения всем соседям. В зависимости от эффективности выбранного lookahead, нулевых сообщений может быть на порядок больше информационных

Во-вторых, выбор самого lookahead зачастую становится проблемой. В крайних случаях, исследователи просто полагают его равным 1 ед. модельного времени (или некоторому ∆t).

Что касается оптимистических алгоритмов, то следует отметить, что механизм мгновенных снимков требует огромное количество памяти, поскольку логический процесс заведомо *не располагает* информацией о том, будет совершён откат или нет. Это приводит к тому, что требуется сохранять снимок модели после *каждого* обработанного события. Как следствие, память следует каким-то образом освобождать. Для этих целей авторы придумали понятие *GVT*.

Определение. *GVT* (от англ. «Global Virtual Time» – глобальное виртуальное время) – нижняя граница временных меток всех событий на всех логических процессах и всех сообщений в системе с учётом всех рекурсивных откатов.

Таким образом, если локальное время процесса t=60, а GVT=15, то все моментальные снимки модели для t<15 гарантированно не будут востребованы, а значит, их можно удалить. Следует подчеркнуть, что GVT также служит надёжным средством фиксации операций ввода-вывода, которые, очевидно, невозможно «откатить» (так называемые *Irrevocable operations*).

Обычно логические процессы считают собственную оценку GVT (которая, как легко заметить, будет ниже или равна реальному значению). Вычислив оценку GVT, процесс фиксирует операции ввода-вывода и освобождает память. Данный процесс в западной литературе получил название *Fossil Collection* (букв. – «сбор ископаемых»).

Напоследок отметим, что глубокие рекурсивные откаты представляют наибольшую угрозу производительности вычислительного процесса, ведь в этом случае алгоритм проигрывает не только по памяти и процессорному времени локального узла, но и становится причиной излишнего расхода памяти и процессорного времени на других узлах. Если процессы на взаимные откаты и антисообщения тратят времени больше, чем на реальные вычисления, то это несомненно ставит под угрозу общую эффективность моделирования.

Общая сравнительная характеристика алгоритмов синхронизации

Ниже в таблица 1 представлена сравнительная характеристика консервативных и оптимистических алгоритмов синхронизации. Обращаем внимание: сравниваются два классических алгоритма (Null Messages Algorithm и Time Warp) без каких-либо модификаций.

Таблица . Сравнительная характеристика алгоритмов синхронизации

| **Признак сравнения** | **Null Messages Algorithm** | **Time Warp Algorithm** |
| --- | --- | --- |
| Требует пересылки сообщений по неубыванию временных меток | Да | Нет |
| Требует фиксированную топологию сети | Да | Нет |
| Требует, чтобы коммуникационная среда гарантировала доставку сообщений в порядке их отправления | Да | Нет |
| Требует механизм разрешения тупиков | Да | Нет |
| Требует наличия служебных сообщений | Да (нулевые сообщения) | Да (Антисообщения) |
| Дополнительные вычисления | LBTS | GVT |
| Требует механизм откатов | Нет | Да |
| Уровень параллелизма | Низкий | Высокий |
| Требования к памяти | Расходы памяти несущественны | Расходы памяти существенны |
| Расход времени процессора | Время может тратиться на ожидание (простой процессора) | Время может тратиться на ненужные вычисления (ввиду откатов) |
| Эффективность вычислений | Зависит от времени ожидания | Зависит от количества откатов |
| Сложность реализации | Относительно проста | Весьма не тривиальна |

Поскольку оптимистические алгоритмы всё же являются более перспективными в плане параллелизма, большинство специалистов в области имитационного моделирования ставят объектом исследования именно их. Была предложена масса улучшений и модификаций существующего Time Warp, направленных на устранение его многочисленных недостатков.

Данная работа также ставит целью разработать оптимистический алгоритм, который был бы эффективнее за счёт использования знаний о модели.

1. Оптимизация алгоритмов синхронизации логических процессов

В данной главе будут предложены улучшения, оптимизации и модификации стандартных алгоритмов синхронизации логических процессов. Что касается оптимистических методов, то помимо улучшения непосредственно самих алгоритмов, авторы пытаются устранить побочные эффекты «оптимизма».

Модификации консервативных алгоритмов

Lookahead

Первой модификацией алгоритма было использование уже описанного ранее lookahead. Изобретателями данного приёма являются Nicol и Rilay (1988г.).

Данный механизм является одним из самых эффективных, поскольку он использует знания о модели.

Carrier-Null Message

Алгоритм Carrier-Null Message был изобретён британскими исследователями Wengton Cai и Stephan Turner[] в 1990г. и позже обобщён соотечественником Kennet Wood в 1992г.

[…]

Control flow graph

Авторами данного подхода являются Cota и Sargent (1990г.)

[…]

Lin, Lazowska и Baer в 1990г. опубликовали статью, где был предложен класс систем, для которых теоретически невозможно предсказать значение lookahead.

Deadlock detection and recovery algorithm

Авторами данного алгоритма по-прежнему выступают Chandy и Misra.

[…]

Bounded lag simulation protocol

Иное название данного подхода – алгоритм консервативного временного окна (Conservative Time Window Algorithm). Автор протокола – Лубачевский (1989г.)[]

Conditional Events algorithm

Алгоритм условных событий был разработан исследователем Sheirman в 1996г [].

Dag Consistent Parallel Simulation

Авторы – Lim, Low и Turner

Прочие алгоритмы

Kumar (1989г.)

Nevison (1990г.)

Wriese (1990г.)

Модификации оптимистических алгоритмов

Lazy Cancellation Algorithm

Алгоритм Джефферсона фактически использует механизм «принудительной отмены» (Aggressive Cancellation). Anat Gafny в 1988г. предложил механизм так называемой «отложенной отмены» (Lazy Cancellation).

Adaptive Cancellation Algorithm

Автор – Wilsey.

Jump Forward Algorithm

Метод также носит название «Алгоритм отложенного отката» (Lazy reevaluation или Lazy rollback algorithm). Автор – Fugjimoto (1990г.) []

Wolf Calls Algorithm

Madissetti, Warland и Messerschmiht в 1988г. [] предложили алгоритм «Волчьего воя».

Moving Time Windows

Алгоритм, предложенный авторами Sokol, Brisco и Wailend в 1988г., до сих пор остаётся самой популярной идеей борьбы с чрезмерным «оптимизмом» [].

Filtered Rollback Algorithm

Идея, предложенная Swartz и Weitson в 1989г. [], фактически является объединением оптимистического Time Warp и консервативного Bounded Lag алгоритмов.

Direct Cancellation Algorithm

Механизм «Прямой Отмены» впервые ввёл Fujimoto (1989г.)[] для архитектур с общей памятью.

Space-Time Simulation

Chandy и Sheirman в 1992г. [] предложили довольно необычную, но весьма эффективную идею…

Filtering Algorithm

Prakash и Subramanian (1992г.) []

Breathing Time Buckets

Предложенный автором Steinman в 1993г. [] алгоритм Breathing Time Buckets и его модификация Breathing Time Warp (комбинация Breathing Time Buckets и Time Warp) получили весьма широкое распространение за счёт полного отказа от анти-сообщений.

Rollback Relaxation Algorithm

Palaniswamy, Aji

Genetic Algorithm for Moving Time Windows

Wang, Tropper

Probabilistic Time Window Adaptation

Luthi

Synchronization based on FIPA-Protocol

Pawlaszczyk

Прочие алгоритмы

Алгоритм, использующий LookBack (Chen, Szymanski)

Алгоритм, использующий оценку GVT (сообщение не отправляется, пока GVT не вырастет до его метки)

Методы, направленные на устранение побочных эффектов оптимистических алгоритмов

Алгоритм Pruneback (Preiss & Loucks)

метод освобождения памяти

Алгоритм Adaptive Checkpointing (Fleischmann)

cost function to balance cost of saving states VS. cost of coasting forward over unsaved states

Алгоритм Reverse Computation (Carothers)

Алгоритмы эффективного вычисления GVT

1. Алгоритмы синхронизации агентов, основанные на знаниях о модели

Данная глава содержит описание и результаты разработки новых алгоритмов синхронизации, использующих знания об агентной модели. Алгоритмы относятся к классу оптимистических. В этой главе будем полагать, что знания о модели известны априори и полном объёме. Позже в главе 6 будет продемонстрировано, как можно извлекать недостающие знания в ходе имитационного прогона.

Ограничения и сфера применимости алгоритмов

Итак, как и любого алгоритма, сначала следует определить предмет исследования, цель, существующую проблему, для решения которой разрабатывается алгоритм, а также ограничить те условия, в рамках которых алгоритм может быть применим.

Далее по тексту работы предлагаемое семейство алгоритмов будет носить внутреннее наименование KBASA (от англ. Knowledge Based Agent Synchronization Algorithms).

Мы рассматриваем класс параллельных дискретно-событийных систем агентного моделирования (Agent based parallel discrete event simulation). Поскольку KBASA является оптимистическим алгоритмом, для него справедливы утверждения, принятые в классическом Time Warp:

1. топология сети логических процессов может быть произвольной (и может меняться в ходе имитационного прогона);
2. коммуникационная среда должна гарантировать доставку сообщений, но при этом допускается ситуация, когда сообщения приходят не в том порядке, в котором были отосланы

Условие семантической определённости

Теперь сформулируем основное условие ограничения для предлагаемого алгоритма.

Утверждение (*условие семантической определённости*). Для корректной работы алгоритмов KBASA требуется:

где e – событие (e ∈ Events)

agens, patiens – агенты (agens, patiens ∈ Agents)

S – множество состояний модели

f – предикат, преобразующий модель из одного согласованного непротиворечивого состояния s ∈ S в другое

Иными словами, для каждого события в любой момент времени можно выделить тройку {агенс-пациенс-предикат}.

* Агенс – активный участник коммуникации – агент, который выполняет действие над другим агентом.
* Пациенс – участник коммуникации, над которым производится действие.
* Предикат – действие, выполняемое пациенсом по инициативе агенса.

*Примечание. Термины* ***агенс*** *и* ***пациенс*** *были взяты из лингвистики (раздел «семантическая (актантная) структура предложения»). В языкознании чаще применяют термины «агент» и «реципиент», однако в силу перегруженности термина «агент» (агент как интеллектуальная программная сущность и агент как семантическая роль участника ситуации) было принято решение оставить вариант в оригинальной трактовке.*

Насколько данное условие сокращает круг моделируемых задач? На самом деле, для большинства систем имитационного моделирования вполне можно выделить тройку {агенс-пациенс-предикат} для каждого события в модели. Более того, в данной работе рассматриваются агент-ориентированные модели, а для такого рода систем условие семантической определённости вполне естественна.

Гипотеза. Для любой системы распределённого агентного имитационного моделирования выполняется условие семантической определённости.

Приведённое выше утверждение сформулировано в виде гипотезу, поскольку, как было упомянуто в главе 1, точного определения агентной системы не существует.

Цели алгоритмов

Многочисленные модификации алгоритма Time Warp используют различные подходы к устранению его недостатков: сдерживание оптимизма, борьба с откатами, борьба с каскадными откатами, новые методы вычисления GVT, механизмы управления памятью, уменьшение накладных расходов, снижение нагрузки на коммуникационную среду и т.д.

Целью предлагаемых алгоритмов является уменьшение количества откатов и, как следствие, суммарного времени моделирования.

Примечание: далее по тексту термины «сообщение» и «событие» могут употребляться как синонимы (что означает: сообщение, несущее информацию о событии).

Природа откатов

К сожалению, в литературе практически нет упоминаний о причинах возникновения и классификации откатов, особенно для агентных систем. В чём же заключается природа тех самых откатов, которые мешают оптимистическим алгоритмам стать де-факто стандартным решением проблем синхронизации?

Чтобы предложить эффективные методы борьбы с откатами, давайте сначала попробуем разобраться в том, что это такое, и вследствие чего они возникают. Напомним, что мы до сих пор ограничиваемся лишь агентными платформами, для которых справедливо условие семантической определённости.

Классификация откатов

Прежде всего, введём несколько определений.

Определение. Событие будем считать *терминальным*, если его обработка не порождает новых событий (от англ. Terminal vertex – концевая вершина графа).

Определение. *Цепочкой отката* называют последовательность событий (и/или состояний модели), упорядоченную по убыванию временных меток (т.е. в обратном порядке).

***► По числу вовлечённых логических процессов***

* Локальные.
* Каскадные.

Локальный откат затрагивает лишь текущий логический процесс, каскадный – два или более (см. определение в главе 2).Очевидно, последние представляют наибóльшую угрозу. К настоящему времени уже разработаны алгоритмы, нейтрализующие каскадные откаты, например, Breathing Time Buckets (см. главу 3). Основной принцип избавления от каскадных откатов – сдерживание оптимизма и отказ от антисообщений.

***► По коммуникативной цели агента***

* Откаты, вызванные запросом к агенту.
* Откаты, вызванные ответом на запрос.
* Откаты, вызванные событиями, не требующих ответа.

Данная классификация относится, скорее, к категоризации событий, нежели непосредственно самих откатов.

Суть первых двух, как следует из названия, кроется в природе коммуникации агентов. Дело в том, что агенты, в отличие от других моделируемых сущностей, активно общаются друг с другом, а значит, неизбежна ситуация, когда один участник диалога отправляет запрос, а затем ждёт ответа. В этом случае и запрос, и ответ могут привести к откатам.

В последнем случае один агент отправляет сообщение другому и не ждёт ответа. В таком случае говорят, что действие одного агента явилось причиной действий другого (например, охранник открывает дверь банка – тем самым он оповещает ожидающих клиентов о том, что они могут войти). Это пример отката без фактической коммуникации.

***► По наличию парадокса времени***

* Откаты, вызывающие парадокс времени.
* Откаты, не вызывающие парадокса времени.

Парадокс времени заключается в том, что поступившее «из прошлого» событие могло каким-либо образом повлиять на ход моделирования. Это касается не только текущего агента, но и всех участников моделирования. Подобное событие может, например, изменить внутреннее состояния другого агента. Например, покупатель в супермаркете, увидев, как другой покупатель смотрит на дату изготовления молока (да ещё и после этого кладёт пакет обратно), скорее всего также проверит дату изготовления.

К событиям, не вызывающим парадокс времени, относятся те, которые не влияют ход моделирования (имеется в виду промежуток от временнóй метки события до текущего времени процесса). В жизни такие явления встречаются редко, но в имитационных моделях обычно многими несущественными моментами пренебрегают.

***► По числу упущенных событий***

* Откаты, вызванные терминальными событиями.
* Откаты, вызванные нетерминальными событиями.

Откаты, вызванные терминальными событиями, вставляют лишь одно событие в цепочку отката, в то время как нетерминальные события могут вставить 2 или более события.

Алгоритм#1

Итак, исследовав природу откатов, можно приступать к разработке механизмов по их устранению. Для каждого алгоритма требуется, чтобы в распределённой дискретно-событийной системе моделирования выполнялось условие семантической определённости.

Данный алгоритм служит простейшей эвристикой по устранению нежелательных откатов. Позже этот алгоритм будет объединён с алгоритмом#3.

Утверждение. Терминальные события допускаются к обработке «в прошлом», поскольку они не влияют на внутреннее состояние других агентов и не порождают новых событий.

Доказательство. Гарантия того, что не будут сгенерированы новые события, вытекает из определения (см. предыдущий параграф). Факт того, что событие не изменяет внутреннее состояние других агентов, подтверждается правилами дискретно-событийного моделирования: обрабатывая событие, агент изменяет только своё внутреннее состояние. Воздействие на других агентов происходит только путём планирования событий и/или передачей сообщений.

Итак, алгоритм представляет собой всего лишь 1 условный оператор:

**if** (event.t < currentTime) //needs rollback

**if** (event isTerminal)

handleEvent(event)

**else** rollback(event)

Приведённый выше псевдокод является упрощением: в нём не отражена обработка антисообщений, постановка события во входную очередь и т.д. Так, например, в коде указывается строгий знак сравнения, что справедливо, в общем случае, лишь для «позитивных» сообщений. Для антисообщений знак должен быть нестрогим, чтобы откатились все события, включая текущее.

Хорошим примером терминального события могут служить сообщения для специального агента, хранящего общее состояние (shared state) модели.

Алгоритм#2

Данный алгоритм относится к классу эвристических методов, сдерживающих оптимизм. К таким, например, относят алгоритмы Breathing Time Buckets и Moving Time Windows. В основу алгоритма положена идея коммуникации агентов. Если системе известны знания о том, какие формы коммуникации существуют в модели, эти знания с успехом можно использовать для предотвращения откатов.

Утверждение. Если агенту приходит сообщение с запросом, и системе достоверно известно, что оно требует ответа, то разнесение агентов по разным логическим процессам чревато многочисленными откатами.

Из данного утверждения можно сделать 2 предположения в пользу решения проблемы:

* выполнить кластеризацию агентов и разместить тесно связанные агенты на одном логическом процессе; кластеризация и балансировка может быть выполнена как статически до запуска модели, так и динамически в ходе имитационного прогона
* при обработке события, инициирующего коммуникацию, блокировать текущий логический процесс до момента прихода ответа от другого участника коммуникации

Алгоритм был разработан в поддержку 2-го решения, однако в общестратегическом плане наиболее перспективным вариантом окажется интеграция обоих подходов.

Итак, алгоритм выглядит следующим образом:

Логический процесс при обработке очередного события обнаруживает, что оно инициирует процесс коммуникации.

Определение. Событие называется *блокирующим*, если для него выполняются 2 условия:

* событие подлежит отправке на другой логический процесс;
* событие содержит запрос, требующий ответа со стороны другого участника коммуникации.

Событие, содержащие ответ на блокирующее событие, будем называть *разблокирующим*. Совокупность блокирующего и разблокирующего событий назовём *контрарной парой*.

При обработке блокирующего события процесс останавливает работу до тех пор, пока он не получит ответа на запрос. Во время блокирования *никакие* другие события, в т.ч. запланированные для других агентов, не выполняются.

Знания о том, какие события являются блокирующими, поступают в систему извне в виде некоторой спецификации модели (онтологии).

Проблема взаимной блокировки

Главной проблемой, как выяснилось из экспериментов (см. главу 7), является возможность появления тупиковых ситуаций (deadlocks). Действительно, ввиду стохастического поведения логических процессов возможна такая ситуация, когда оба процесса практически одновременно обрабатывают блокирующие события, отправляют запросы друг другу и ждут друг от друга ответа. В случае 3-х или более процессов может образоваться цикл (рис.)

[рис.]

Для разрешения тупиковых ситуаций существует множество методов и механизмов. В реализованной программной системе механизм выглядит следующим образом:

Логический процесс, сталкиваясь с блокирующим событием, обрабатывает его, высылает сообщение, блокируется и отсылает ещё одно сообщение специального типа (LockRequest) на тот же узел, на который отправлено первое сообщение.

Любой процесс, получивший LockRequest, должен немедленно выслать обратно сообщение специального типа LockResponse. Это говорит о том, что процесс уже заблокирован, а значит, он не сможет выслать разблокирующее событие.

***Замечание к реализации***

Для механизма выборки очередного события и механизма обработки сообщений LockRequest обязательно нужно разрешить проблему взаимного исключения. В противном случае может оказаться, что логический процесс выберет блокирующее событие, остановит работу, но на запрос TimeRequest он не вышлет ответа TimeResponse.

Проблема бесконечного ожидания

Другой проблемой при реализации данного алгоритма служит появление преждевременного отката. Эта явление настолько неочевидно, что выявилось лишь только на этапе тестирования и эксплуатации разрабатываемой агентной платформы.

Общая схема возникновения бесконечного ожидания проиллюстрирована на рис.

[рис.]

Продемонстрируем небольшой пример. Пусть логический процесс LP1 обработал цепочку событий:

Отсюда локальное время LP1 – t=8. Пусть событие E (на процессе LP1) и F (на процессе LP2) образуют контрарную пару. Также положим, что обработка событий D и Е сопровождалась отправкой сообщения на LP2.

Итак, процесс LP1 обрабатывает событие E, в результате чего он блокируется и высылает LockRequest на процесс LP2. Теперь предположим, что 3-й процесс LP3 присылает на LP1 сообщение:

,

которое, очевидно, вызывает откат процесса LP1 до состояния t=4. В результате отката формируются антисообщения для событий D и E, которые отменяет высланные ранее на LP2 сообщения. Предположим, что LP2 не успел обработать событие E, а значит, оно будет взаимно уничтожено с антисообщением.

***Результат***: логический процесс LP2 yf *даже не узнает* о том, что его сосед ждёт разблокирующего события F. В итоге LP1 заблокирован навсегда.

Существует несколько путей выхода из сложившейся ситуации:

* отменять блокировку по отсылке антисообщения
* процессу-получателю блокирующего события просматривать буфер входных сообщений, чтобы впоследствии предвидеть подобного рода ситуации
* дождаться разблокирования, после чего выполнить откат

Как будет показано позднее, первый вариант окажется наиболее прогрессивным, так как он удачно сочетается с алгоритмом#3.

Алгоритм#3

Данный алгоритм также относится к классу эвристических методов, основанных на знаниях о модели, но на сей раз алгоритм уже не сдерживает оптимизм. Алгоритм основан на зависимости/независимости событий и призван, прежде всего, помочь устранить те откаты, которые не являются следствием вопросно-ответного диалога, а значит, не могут быть устранены Алгоритмом#2.

Пусть имеется распределённая дискретно-событийная система агентного моделирования, для которой выполняется условие семантической определённости. Введём отношение *каузальной зависимости* событий.

Определение. Будем говорить, что событие А *каузально зависит* от события В, если агент, планирующий событие А, может изменить планирование любого другого события в будущем под влиянием события В.

Приведём простой пример. Если автомобиль переезжает железнодорожный переезд на открытый шлагбаум (событие – шлагбаум открыт), то, скорее всего, все будущие события не изменятся, т.е. останутся с прежними временными метками. Если же шлагбаум закрыт, то время всех следующих событий однозначно будет изменено (и в результате, возможно, некоторые события могут быть не запланированы вообще).

Необходимое условие каузальной зависимости

Напомним, что для каждого события можно однозначно выделить тройку {агенс-пациенс-предикат}. Рассмотрим теперь, как можно формально определить каузальную зависимость событий.

Утверждение (*необходимое условие каузальной зависимости*). Для того, чтобы события и были каузально зависимы, необходимо, чтобы *patiens1* = *patiens2*.

Прежде чем доказать это утверждение, покажем сначала справедливость леммы:

Лемма. Необходимое условие каузальной зависимости выполняется для любого терминального события А.

Доказательство леммы. Рассмотрим множество всех произвольных подмножеств событий логического процесса, упорядоченное по возрастанию временных меток (в хронологическом порядке). Обозначим tmin – минимум всех временных меток. Пусть на логический процесс приходит терминальное сообщение, содержащее событие , причём его временнáя метка:

(\*)

Рассмотрим 4 случая:

1. Предположим, существует событие , такое, что agensA = agensB. Такое событие невозможно, поскольку агент, представляющий agensA, располагается на другом логическом процессе, а значит, он не может планировать события на локальном процессе.

2. Пусть существует событие , такое, что agensA = patiensC. Это значит, что существует событие , такое, что , причём . Иными словами, найдётся агент, который спланировал событие C в прошлом. Обработка события А не может повлиять на исход события , т.к. они выполняются разными агентами, а patiensA не планирует новых событий в силу терминальности. Значит, события А и С каузально независимы.

3. Пусть существует событие , такое, что patiensA = patiensD. Событие А обрабатывает агент patiensA, причём в ходе обработки могло измениться его внутреннее состояние. В силу предположения, событие D также обрабатывает агент patiensA, а раз t(A) ≤ t(D), то обработка D, возможно, выполнилась бы иначе (в зависимости от результата обработки события А). Вывод: данный случай действительно допускает возможность каузальной зависимости.

4. Предположим, существует событие , такое, что patiensA = agensE. Раз локальный процесс не имел информации о событии А, то существует событие , такое, что , причём . Т.е. найдётся агент, который спланировал данное событие E в прошлом. Заметим, что выполняется равенство:

Поэтому при получаем случай, аналогичный случаю 3 (если положить ), а вариант не рассматриваем в силу условия (\*).

Таким образом, мы показали, что если события каузально зависимы, то условия 1,2,4 не могут иметь место, а значит, справедлив только 3-й случай (пациенсы событий равны). Условие достаточности доказано.

Доказательство утверждения. Пусть теперь событие А не является терминальным. Значит, существует некоторое событие , такое, что patiensA = agensX (событие X порождено агентом patiensA при обработке события A). Очевидно, X и A каузально зависимы по определению.

Рассмотрим множество событий

Иными словами, событие X будем сравнивать только с событиями из будущего. В итоге получаем новое множество:

Теперь вновь рассмотрим 4 случая, только на этот раз относительно пары события Х и A:

1. Пусть существует событие , такое, что patiensX = patiensF. В ходе доказательства леммы выяснилось, что если пациенсы совпадают, то события казуально зависимы.

2. Пусть существует событие , такое, что agensX = patiensG. Значит, имеет место равенство:

По лемме вновь получаем, что пациенсы равны, а значит вся тройка событий {A, X, G} каузально зависима.

3. Пусть существует событие , такое, что patiensX = agensH. Значит, существует событие , такое, что , причём . Легко видеть ,что выполняется равенство:

А этот случай полностью аналогичен ситуации, рассмотренной в п.1 (если положить )

4. Пусть существует событие , такое, что agensX = agensI. Данный случай в лемме не рассматривался, потому что мы имели дело с удалённым агентом. Теперь же агент локальный, поэтому случай следует рассмотреть. Из предположения следует, что существует событие , такое, что , причём . В итоге:

Этот случай полностью аналогичен п.2 (если положить ).

Таким образом, мы показали, что если события каузально зависимы, независимо от того, терминальные они или нет, то их пациенсы равны. Условие достаточности доказано.

***Замечание***

Доказанное условие является необходимым, но не является достаточным. Другими словами, обратное не всегда верно: если пациенсы двух событий равны, то не факт, что они каузально зависимы.

Описание алгоритма

Алгоритм основан на принципе учёта казуальной зависимости. Основной его принцип заключается в анализе зависимостей внутри цепочки событий с целью принятия решения о том, следует ли выполнить откат или нет.

Суть оптимистических алгоритмов заключается в том, чтобы при получении сообщения с временнóй меткой, меньшей текущего времени (straggling message), выполнить откат на прежнее непротиворечивое состояние. Делается это для того, чтобы избежать парадокса времени. Возникает вопрос, а всегда ли подобное сообщение вызывает парадоксы времени?

Хотелось бы найти такое эффективное решение, которое без наличия специального алгоритма отслеживания сообщений «из прошлого» не создавало бы нежелательные откаты, но при этом устраняло проблему временных парадоксов.

Данная формулировка применяется в ТРИЗе для разрешения технических противоречий. Алгоритм#3 делает первый шаг на пути к этой цели.

Итак, общая идея алгоритма: при получении сообщения «из прошлого» логический процесс вместо немедленного отката выполняет анализ всех подлежащих откату событий на предмет каузальной зависимости.

Пусть на логическом процессе LP1 обработана цепочка событий . Далее процесс LP2 высылает событие . Очевидно, классический оптимистический алгоритм выполнит откат до состояния, заданным событием *B*. Цепочка отката примет следующий вид:

В нашем алгоритме выполняется поиск события, каузально зависимого от . по всем событиям из цепочки отката. Если хотя бы одно из этих событий каузально зависимо – алгоритм завершает свою работу с ответом *false* (обработка события . запрещена, нужно выполнить откат).

Если бы сообщение . было терминальным, а все события из цепочки отката были от него независимы, то алгоритм бы завершил работу с ответом *true* (обработка события . разрешена, т.к. она не вызывает парадоксов времени).

Рекурсивная обработка

Если событие не является терминальным, то на этом алгоритм не останавливается, поскольку обработка события может повлечь планирование новых событий, которые могут поместиться внутри цепочки откатов. Поэтому следующим шагом является так называемая «псевдообработка» события.

На данном этапе событие обрабатывается «виртуальным» симулятором. Обработка не фиксируется как реальный процесс моделирования: время не увеличивается, состояние модели не меняется, статистика не собирается. Единственная цель – получить на выходе набор новых событий, планируемых агентом. Пусть в нашем примере событие . генерирует цепочку .

Далее все новые «псевдособытия» следует отфильтровать. Те из них, чьи временные метки больше или равны текущему времени процесса (t=12), исключаем из рассмотрения. В нашем примере остаются . Далее алгоритм повторяется рекурсивно для каждого из оставшихся «псевдособытий».

Следует отметить, что на каждом шаге рекурсии текущее событие проверяется на каузальную зависимость только с событиями из будущего. Т.е. следует проверять с парой , а – только с . Если проверка прошла успешно (события независимы), то для и снова проводится «псевдообработка», и процесс повторяется до тех пор, пока в цепочке отката не останется новых событий.

Последний шаг алгоритма – свёртка. Если для события . и для всех его потомков в текущей цепочке отката нет ни одного каузально зависимого события –алгоритм завершает работу с ответом *true* (обработка события безопасна). В противном случае, алгоритм завершает работу с ответом *false*.

Основное замечание к вышесказанному – проверять наличие каузальной зависимости между событиями достаточно сложно и не всегда возможно за приемлемое время. А вот проверять отсутствие этой зависимости уже проще – достаточно воспользоваться *необходимым условием каузальной зависимости*, которое было доказано параграфом выше.

Общий алгоритм обработчика сообщений

Прежде чем переходить к теоретической оценке сложности, предложим общий алгоритм цикла обработки сообщений на псевдокоде. Ниже показана обработка принятого сообщения:

**final def** handleMessage(m: BaseMessage) {

// проверка на Rollback

**if** (getTime > m.t) m **match** {

**case** em: EventMessage =>

**if** (!isOK(em.timeevent)) rollback(m)

**case** \_: AntiMessage => rollback(m)

}

// постановка во входную очередь

m **match** {

**case** \_: AntiMessage => inputQueue find {\_ == m} map {\_ =>

inputQueue -= m

}

**case** em: EventMessage =>

inputQueue = (inputQueue += em).sorted

checkUpMessage(em)

}

}

Функция выполняет 2 этапа. На первом этапе алгоритм выполняет откат, если сообщение пришло «из прошлого». Для антисообщений откат обязателен, для «позитивных» сообщений – только если алгоритм#3 посчитает обработку события опасной (и вернёт *false*). На втором этапе сообщение помещается в буфер, если оно «позитивное», либо взаимно уничтожается с одним из существующих в буфере, если это антисообщение.

К приведённой выше функции следует добавить небольшое, но важное замечание. Проверка

**if** (getTime > m.t)

в действительности справедлива только для «позитивных» сообщений. Если мы имеем дело с антисообщениями, то знак *обязательно* должен быть нестрогим. Это означает, что другой логический процесс желает отменить сообщение, которое локальный процесс вот-вот обработал. Несоблюдение этого условия приведёт к следующему:

* откат не будет выполнен,
* антисообщение не уничтожит «позитивное» сообщение во входном буфере (ввиду отсутствия последнего).

Этот случай, очевидно, нарушает локальную каузальность. В программной реализации этот момент учтён, но в приведённом фрагменте кода опущен в целях наглядности.

Примечание. Выражение

find {\_ == m}

выполняет поиск сообщения, равного антисообщению **m**. Операция равенства введена искусственно таким образом, что «позитивное» сообщение равно антисообщению.

Теперь приведём код функции «*isOK*». Фактически, она является точкой входа в рекурсию.

**final def** isOK(e: TimeEvent): Boolean =

**if** (isSafe(e)) runPseudoEvent(e)

**else** **false**

Ниже приводится фрагмент рекурсивной функции «*isSafe*»:

**private def** isSafe(e: TimeEvent): Boolean = {

// если событие "stateless", то оно априори безопасно

**if** (Knowledge isStateless e.event) **true**

**else** {

// просматриваем стек состояний (в направлении "в прошлое")

**var** result = **true**

**var** storage: List[(TimeEvent, Array[Byte])] = Nil

**var** q = stateStack peek()

**while** (result && q != **null**)

q = **if** (q.\_1.t < e.t) **null**

**else** {

**if** (isLocal(q.\_1.event) && correlate(e.event, q.\_1.event))

result = **false**

storage ::= stateStack pop()

stateStack peek()

}

// заполняем стек обратно

**for** {a <- storage}

stateStack push a

result

}

}

Поясним представленный фрагмент. Функция «*isStateless*» представляет собой Алгоритм#1. Если событие обладает данным свойством, то его обработка считается безопасной. Если же это не так, то оптимистично предполагаем, что событие безопасно и начинаем итерационно просматривать стек состояния модели.

На каждом шаге алгоритм изымает событие из стека, проверяет, не является ли оно каузально зависимым с текущим событием, и кладёт событие во вспомогательное хранилище (storage). Ниже приводится фрагмент функции «correlate», основанная на доказанном выше *необходимом условии каузальной зависимости*.

**private def** correlate(cur: AgentEvent, that: AgentEvent): Boolean = cur.patiens == that.patiens

Если функция «*corellate*» вернёт ***false***, то мы достоверно можем быть уверены в том, что события независимы. А значит, алгоритм продолжает работу со стеком состояний. В противном случае функция говорит о теоретической возможности существования каузальной зависимости, а значит, процедуру нужно остановить с ответом ***false*** (событие небезопасно).

Обратите внимание, что проверку «correlate» нужно делать лишь в том случае, если проверяемое событие адресовано локальному процессу (функция «*isLocal*»). Итерационная работа со стеком состояний заканчивается в одном из трёх случаев:

* найдено событие из цепочки отката, такое, что оно каузально зависимо с проверяемым событием (алгоритм возвращает ***false***)
* стек пуст (алгоритм возвращает ***true***)
* пройдена вся цепочка отката, т.е. проверены все события, временнáя метка которых больше проверяемого события (алгоритм возвращает ***true***)

Перед выходом из функции следует не забыть восстановить стек в предыдущее состояние (по принципу «Ханойской башни»).

Ну и наконец, представим функцию «псевдообработки» события, которая рекурсивно вызывает функцию *isOK*.

**private def** runPseudoEvent(e: TimeEvent): Boolean = {

**val** lst = **for** {

event <- simulateStep(e) **if** event.t < getTime

} **yield** isOK(event)

lst forall {b => b}

}

Здесь функция «*simulateStep*» возвращает список событий-потомков, т.е. тех, которые будут (или могут быть) сгенерированы обработкой текущего события. Обратите внимание – список фильтруется в соответствии с условием, что временнáя метка события меньше текущего времени логического процесса (функция «*getTime*»).

Далее для каждого события рекурсивно вызывается функция «*isOK*», и вычисления повторяются с самого начала. Если список, полученный на выходе функции «simulateStap» и отфильтрованный в соответствии с текущим временем, окажется пустым, то рекурсия останавливается.

Оператор ***yield*** применяет некоторую функцию и обратно упаковывает тот список, который распаковал оператор ***for*** (данная конструкция называется *for-comprehension*). Таким образом, в переменной «*lst*» будет находится список булевских значений, показывающих, безопасно ли то или иное событие. Выражение

*lst forall {b => b}*

возвращает ***true***, если все элементы списка равны ***true*** (либо список пуст).

Теоретическая оценка сложности

Приведём теоретическую оценку сложности Алгоритма#3, объединённого с Алгоритмом#1, по процессорному времени (оценка сложности Алгоритма#2 не приводится в силу тривиальности).

Сначала некоторые формальные допущения.

* Арифметические и логические операции, операции сравнения, операции присваивания, операции работы над стеком (*push*, *pop*, *peek*) занимают константное время (). Мы будем их обозначать просто константой «1»
* Операция вставки в список производится в начало списка. В большинстве реализаций (в т.ч. в языке Scala) операция занимает константное время (). Мы также будем использовать константу «1»
* Обращение к подсистеме знаний занимает константное время
* Обращение к функции «*isLocal*» занимает константное время
* Обращение к функции «*correlate*» в соответствии с *необходимым условием каузальной зависимости* сводится к операции сравнения, а значит, занимает константное время
* Обозначим Inotsafe – индикатор того, что событие небезопасно
* Обозначим Inotstateless – индикатор того, что событие не обладает свойством stateless.
* Обозначим It<cur\_t – индикатор того, что событие имеет временную метку меньше, чем текущее время процесса
* Пусть {n1, n2, …, np} – числовая последовательность, показывающая размер цепочек отката на каждом шаге рекурсии
* Пусть {k1, k2, …, kp} – числовая последовательность, составленная из членов последовательности {n}, каждый элемент которой равен:
  + номеру события из цепочки отката, которое является каузально зависимым с проверяемым событием
  + ∞, если все события из цепочки отката независимы от проверяемого события
* Пусть {m1, m2, …, mq} – числовая последовательность, показывающая количество сгенерированных сообщений-потомков на каждом шаге рекурсии.

Итак, функция «*isOK*» состоит из сложности функций «*isSafe*» и «*runPseudoEvent*».

Оценим каждое слагаемое правой части по отдельности. Функция Tsafe учитывает сложность обращения к подсистеме знаний, двух присваиваний, цикла ***while*** и ***ni*** стековых операций *push*.

Условие цикла ***while*** содержит операцию сравнения и логическую операцию. Тело цикла содержит операцию присваивания, операцию сравнения, функции «isLocal» и «correlate», логическую операцию, две операции над стеком и вставку в список. Всего тело цикла будет выполнено min(ki, ni) раз.

Подставляя выражение для Twhile в Tsafe, получаем:

Наконец, осталось получить оценку для Tpseudoevent. Здесь учитывается сложность функции «*simulateStep*», ***mi*** сравнений, не более ***mi*** вызовов функций «*isOk*» и операция ***forall***, которая фактически приравнивается к нескольким логическим операциям.

Запишем теперь общее рекуррентное соотношение:

Данную формулу можно считать итоговой. Можно, тем не менее, получить минимальную, максимальную и среднюю оценки сложности.

Для получения минимальной оценки достаточно положить все индикаторы равными нулю (***Inotstatless=Inotsafe=It<cur\_t=0***). В итоге получаем константное время:

Для получения максимальной оценки сложности следует положить все индикаторы, равными единице (***Inotstatless=Inotsafe=It<cur\_t=1***). Тогда формула примет следующий вид:

Для получения средней оценки следует «усреднить» рекурсивные вызовы, числовые последовательности и индикаторы:

где ***τ*** – либо ***p***, либо ***q*** в зависимости от типа индикатора.

Тогда рекуррентное выражение принимает следующий вид:

Перенося ***Tok*** в левую часть, получим:

Следует отметить справедливость следующего выражения:

Данное утверждение строго доказывать не будем, однако отметим тот факт, что именно индикатор ***It<cur\_t*** влияет на выход из рекурсии, поэтому его усреднённая оценка будет достаточно близка к нулю.

Учёт знаний о модели

В предыдущем параграфе были получены теоретические оценки сложности по времени. Следует отметить, что практически все функции и операции занимают константное время O(1), за исключением единственной – . В общем случае, сложность этой функции зависит от особенностей реализации «псевдообработки» события и может быть достаточно большой.

В главе 7 будет показано, как можно использовать знания о модели для получения списка событий-потомков без применения физической «псевдообработки». В этом случае функция обращается к подсистеме знаний, а значит, имеет место равенство:

С учётом этого соотношения, теоретические оценки могут быть переписаны в следующем виде:

***Теоретическая оценка (рекуррентная формула):***

***Минимальная оценка:***

***Максимальная оценка:***

***Средняя оценка:***

Сложности в реализации алгоритмов

Как и предыдущий, данный алгоритм имеет две сложности при реализации на практике. Первая из них касается интеграции с Алгоритмом#2. Проблема заключается в том, что Алгоритм#3 разрешает обработку событий «из прошлого». Однако может оказаться, что Алгоритм#2 заблокировал логический процесс до получения разблокирующего события.

Поскольку Алгоритм#3 гарантирует, что обработка подобного рода событий не вызовет нарушения локальной каузальности, можно смело разрешить их обработку даже в случае заблокированного процесса.

Если первая проблема, вообще говоря, носит оптимизационный характер и в целом не является критичной, то вторая проблема более опасна – во-первых, она не столь очевидна, а во-вторых, в редких случаях она может вызвать парадоксы времени.

Суть проблемы заключается в описанной выше «псевдообработке» события. Согласно алгоритму, после обработки новые события фильтруются по значению временнóй метки. Дело в том, что в обработчике могут встречаться генераторы случайных чисел, а это может привести к тому, что временные метки событий-потомков в «псевдообработке» могут отличаться от реальной обработки.

Приведём небольшой пример. Пусть локальное время процесса t=20. Событие согласно Алгоритму#3 подвергается «псевдообработке», в результате которой генерируется цепочка событий . После фильтрации по временнóй метке остаётся цепочка .

Теперь положим, что алгоритм разрешил обработать событие , но в процессе реальной обработки генераторы случайных чисел сработали так, что сгенерировалась цепочка . Отсюда видно, что событие не было проверено алгоритмом, а значит, гипотетически может вызвать нарушение локальной каузальности.

Предлагается 3 пути решения данной проблемы:

1. «Псевдообработку» приравнять к реальной обработке. В этом случае придётся заводить некоторый отдельный буфер для хранения временных событий-потомков.
2. выполнить разбор генераторов случайных чисел. Известно, что генераторы формируют последовательности случайных чисел на основе специальных функций, учитывающих предыдущие значения. Начальное значение, используемое для генерации первого числа, называется *seed*. Таким образом, в любой момент времени можно воссоздать такую последовательность, зная *seed* и функции преобразования.
3. Использовать методы, основанные на знаниях. Подробнее об этом способе будет рассказано в главе 7. Отметим, что в реализации был выбран именно этот подход.
4. Разработка платформы. Архитектура, технологии и инструментальные средства

В данной главе будет предложена реализация библиотеки синхронизации агентов на основе знаний о модели. Особое внимание будет уделено архитектуре программного комплекса и модели акторов. Поскольку проект является open-source решением, он может свободно изменяться и модифицироваться в любое время. Поэтому отметим, что в этой главе речь идёт о проекте в состоянии на 22 мая 2013г.

Модель акторов

Модель акторов представляет собой математическую модель организации параллельных вычислений в многопроцессорных архитектурах с общей или раздельной памятью.

Первые упоминания о модели акторов имеют место в докторской диссертации американского учёного Iren Graif (MIT, 1975г.), докторской диссертации исследователя William Klinger (MIT, 1981г.) и работе «Законы взаимодействующих параллельных процессов» (авторы G. Baker и K. Hewitt, IFIP, 1977г.). В настоящее время формальное представление модели акторов используется в *теории исчисления процессов*.

Общая суть такова, что все моделируемые сущности многопроцессорных ВС представляются в виде *акторов* (по аналогии с объектами в объектно-ориентированном программировании). Акторы представляют собой некоторое воплощение моделируемого объекта, которое может:

* иметь внутреннюю программную логику;
* иметь локальное состояние;
* отправлять конечное число сообщений другим акторам;
* создать конечное число новых акторов в системе;
* прекратить своё существование (если ему пришлют сообщение специального типа).

С точки зрения реализации в современных мультиядерных архитектурах актор представляет собой самостоятельный, легковесный, асинхронный образ потока управления (thread). На практике один поток обычно содержит десятки или сотни таких акторов.

Модели акторов априори присущ внутренний параллелизм за счёт полной асинхронности операций, выполняемых акторами. Подразумевается, что порядок действий актора является недетерминированным: он может в одно и то же время получить десяток сообщений, обработать часть из них, затем послать ответные сообщения, создать новых акторов и т.д.

В целом считается, что акторы знают адреса всех своих соседей. Однако не исключается динамическое создание акторов, в результате которого остальные соседи получат адрес нового актора, чтобы включить его в общую сеть. Также модель полагает, что сообщения могут приниматься в произвольном порядке (т.е. порядок доставки не гарантируется).

Модель акторов может быть управляемой и неуправляемой. В первом случае система акторов имеет так называемого *супервизора* – специального актора, контролирующего систему и не участвующего в вычислительном процессе. В случае прерывания работы актора супервизор может сохранить его состояние, завершить работу актора и на его месте создать новый актор. В неуправляемых моделях супервизор отсутствует.

Формальное представление модели акторов имеет различные подходы, в частности такие, как:

* Денотационная семантика.
* Операционная семантика.
* Алгебра акторов.

Теоретическая база модели акторов строится на идее нескольких предшествующих моделей вычисления:

* Лямбда-исчисление (теория исчислимых функций Алонзо Чёрча)
* Система Smalltalk-71 (на базе идей языков Smalltalk и Simula)
* Сети Петри

Модель акторов долгое время оставалась теоретической разработкой для многопроцессорных MPP-машин и суперкомпьютеров. В настоящее время популярность данной модели возросла в сотни и тысячи раз в связи массовым распространением параллелизма. Самыми популярными современными языками, использующими модель акторов, является Erlang, Scala и Go. Также существует множество библиотек и фреймворков для других языков.

Напомним, что модель акторов является не просто механизмом передачи сообщений (типа стандарта MPI), а полноценной моделью параллельных вычислений.

Итак, почему же было выбрано решение применять модель акторов? Ответ очевиден – актор является практически полноценным прототипом реактивного агента, способного принимать сообщения и отвечать на запросы. А поскольку мы имеем дело с параллельным дискретно-событийным моделированием (PDES), то модель акторов автоматически определяет модель параллелизма.

Элементы языка Scala

Scala была выбрана в качестве языка программирования для реализации агентной платформы. Данный раздел не ставит своей целью рассказать об общих идеях и концепциях, истоках возникновения, преимуществах и недостатках языка Scala. Здесь будут изложены лишь базовые понятия, которые поспособствуют усвоению дальнейшего материала главы.

В двух словах, Scala представляет собой кросспарадигменный объектно-функциональный язык программирования, совмещающий в себе ООП и функциональное программирование и специализирующийся на создании легкомасштабируемого компонентного ПО.

Scala была создана в 2004г. под руководством Мартина Одерски в Университете EPFL (Lausanne, Switzerland). В настоящее время доступна для платформ Java и .NET Framework. Модель акторов в языке обеспечивается за счёт расширения Akka.

Среди ключевых особенностей языка можно отметить всё то богатство, которое предоставляет ООП и функциональное программирование: единая объектная модель, наличие примесей (traits), техника сопоставления с образцом, лямбда-исчисление, виды (type views), каррирование функций (carrying), Y-комбинаторы, линеаризация типов (type linearization), параметрический и функциональный полиморфизм, вариантность типов, кейс-классы (case classes), вывод типов, поддержка хвостовой рекурсии, наличие многочисленных инструментов по созданию новых языковых конструкций и обработке списков.

В данной главе подразумевается, что читатель знаком и хорошо владеет принципами ООП (на примере языков C++, C#, VB.NET, Objective-C, Java, Delphi). В данном разделе будут коротко представлены некоторые функциональные элементы языка Scala, возможно незнакомые читателю, которые будут применяться в дальнейшем изложении.

Кейс-классы

Кейс-классы (*case classes*) внешне практически ничем не отличаются от стандартных классов, принятых в ООП. Они отличаются по семантике – суть их заключается в том, что они представляют собой некоторую сущность, однозначно задаваемую набором атрибутов.

Приведём простой пример. Пусть имеется класс *Человек*, задаваемый атрибутами *Фамилия*, *Имя*, *Пол*, *Возраст*. Этот класс может выглядеть, например, так:

**class** Person(first: String, last: String, sex: Gender, age: Int) {

...

}

Теперь пусть имеется класс *График\_работы*, задаваемый атрибутами *Будни*, *Суббота*, *Воскресенье*. Класс представим следующим образом:

**case class** Schedule(weekdays: (Int, Int), suturday: (Int, Int), sunday: (Int, Int))

Теперь поставим вопрос: является ли {Иванов Иван, муж, 57} той же самой сущностью, что и {Иванов Иван, муж, 57}? Очевидно, нет – это могут быть разные люди. А является ли график работы {ПН-ПТ: 9-20, СБ: 9-19, ВС: 9-17} равным графику {ПН-ПТ: 9-20, СБ: 9-19, ВС: 9-17}? Семантически это одно и то же.

Среди практических особенностей кейс-классов можно отметить:

* экземпляры могут быть созданы без операции ***new***
* равенство двух экземпляров определяется равенством их атрибутов
* кейс-классы сериализуемы
* автоматически определены методы *toString* и *hashCode*
* экземпляры могут участвовать в сопоставлении с образцом

Во всём остальном это обычные (ссылочные) классы, которые могут содержать локальные переменные и функции. Заметьте, при сравнении учитываются только атрибуты. Локальные переменные не учитываются.

Структуры данных, используемые в системе

К основным типам данных, применяемым в системе, относят события и сообщения.

**Событие агента**

**case class** AgentEvent(agens: *String*, patiens: *String*, predicate: *String*) {

**var** userdata: *Serializable* = None

...

}

Как было упомянуто в главе 4, события должны удовлетворять *условию семантической определённости*, т.е. содержать такие данные, как агенс, пациенс и предикат. Помимо этих трёх полей, класс содержит некоторую произвольную определяемую пользователем информацию ***userdata***. В качестве типа может выступать любой сериализуемый тип данных.

Обратите внимание – ***userdata*** является локальным полем класса. Это значит, что если агенс, пациенс и предикат двух событий равны, то сами события будут считаться равными.

**Событие с временнóй меткой**

**case class** TimeEvent(t: *Float*, event: *AgentEvent*) **extends** Comparable[TimeEvent] {

**def** compareTo(that: *TimeEvent*): *Int* = (math.signum(t - that.t)).asInstanceOf[Int]

...

}

Класс *TimeEvent* представляет собой событие, для которого определена временнáя метка. Для экземпляров данного класса введено отношения порядка, благодаря чему события могут быть отсортированы по времени.

**Сообщение**

**abstract sealed class** Message **extends** Serializable {

**val** sender: *String*

}

*Message* является базовым классом для всех сообщений в системе. Класс несёт в себе лишь информацию об отправителе.

Примечание. Библиотека Akka предоставляет инструменты для определения отправителя сообщения. В нашем случае данные об отправителе оставлены из соображений обратной совместимости.

**Базовое сообщение**

**abstract sealed class** BaseMessage **extends** Message **with** Comparable[BaseMessage] {

**val** id = UUID.randomUUID.toString

**def** t: *Float*

**override def** equals(obj: *Any*) = obj **match** {

case m: *BaseMessage* => id == m.id

case \_ => **false**

}

**def** compareTo(that: *BaseMessage*): *Int* = (math.signum(t - that.t)).asInstanceOf[*Int*]

}

*Базовое сообщение* является суперклассом для сообщений, принятых в области оптимистических алгоритмов синхронизации: информационных («позитивных») сообщений и антисообщений. Базовые сообщения могут быть отсортированы по времени, поскольку они хранят информацию о временнóй метке события. Сама временнáя метка не определена (тело функции ***t*** пусто), поэтому класс является абстрактным.

Обратите внимание на строковое поле ***id***. Оно хранит в себе некоторый 128-битный UUID (*Universally Unique IDentifier* – универсальный уникальный идентификатор), который генерируется на момент создания экземпляра. Это поле требуется для введения отношения равенства сообщений: базовые сообщения равны тогда и только тогда, когда их идентификаторы равны.

Информационное («позитивное») сообщение

**case class** EventMessage(sender: *String*, timeevent: *TimeEvent*) extends *BaseMessage* {

**def** t = timeevent.t

}

Данный класс расширяет базовое сообщение. Это самый главный и самый часто используемый класс сообщений. В области распределённого моделирования – это сообщение, которое несёт в себе информацию о событии. В предыдущих главах было отмечено, что теоретически сообщения и события являются синонимами.

Несмотря на тот факт, что мы имеем дело с кейс-классом, равенство аргументов не приведёт к равенству экземпляров. Это связано с тем, что базовое сообщение переопределяет равенство экземпляров по следующему закону: два объекта равны, если их идентификаторы равны.

**Антисообщение**

**case class** AntiMessage(sender: *String*, eventMessage: *EventMessage*) **extends** *BaseMessage* {

**override val** id = eventMessage.id

**def** t = eventMessage.timeevent.t

}

Данный класс представляет собой антисообщение, созданное на базе информационного сообщения. Класс переопределяет идентификатор так, чтобы он был равен идентификатору «позитивного» сообщения. Это в конечном итоге приведёт к тому, что операция сравнения антисообщения и соответствующего информационного сообщения вернёт ***true***.

**Стартовое сообщение**

**object** StartMessage **extends** Message {

**val** sender = ""

}

*Стартовое сообщение* отправляется процессом-диспетчером всем логическим процессам для обозначения команды запуска моделирования.

**Стоп-сообщение**

**case class** StopMessage(sender: String) **extends** Message

*Стоп сообщение* отправляется процессом-диспетчером всем логическим процессам для обозначения команды остановки моделирования.

**Сообщение со статистикой**

**case class** StatResponse(sender: *String*, statistics: *Statistics*) **extends** Message

Сообщения *StatResponse*, по принятому соглашению, высылают диспетчеру информацию со статистикой, накопленной информационными процедурами, в ответ на сообщение *StopMessage*. Иными словами, логический процесс получает сообщение *StopMessage*, останавливает моделирование и отсылает в ответ данные со статистикой.

**Запрос для определения блокировки процесса**

**case class** LockRequest(sender: *String*) **extends** Message

Подробно назначение данного сообщения было описано в Алгоритме#2: логический процесс, блокируясь, должен убедиться в том, что в системе не возникнет тупика (deadlocks). Поэтому он высылает сообщение *LockRequest*. Если вдруг вернётся ответ в виде сообщения *LockResponse*, то блокировка будет снята.

**Ответ о блокировке процесса**

**case class** LockResponse(sender: *String*) **extends** Message

Данное сообщение высылается логическим процессом в ответ на *LockRequest*, но лишь в том случае, если локальный процесс заблокирован Алгоритмом#2.

**Прочие (служебные) сообщения**

**case class** TimeRequest(sender: *String*) **extends** Message

**case class** TimeResponse(sender: *String*, t: *Float*) **extends** Message

Данные сообщения используются диспетчером для определения условия завершения моделирования по времени.

Многослойная архитектура абстрактного симулятора

Обобщённая структурная схема симулятора изображена на рис.

[рис.]

Как мы видим, архитектура симулятора является не многокомпонентной и не иерархической, как в большинстве других проектов, а многослойной. Достигается это за счёт того, что к модулю, реализующему некоторую базовую функциональность, «примешиваются» другие функциональные блоки, расширяя структуру, поведение и семантику модуля.

В некоторых языках имеет место понятие «примесей» (Mix-in), по факту, представляющих ещё один способ решения проблемы множественного наследования (наряду с интерфейсами в большинстве объектно-ориентированных языках). Примесь – это элемент языка программирования, реализующий некоторое поведение, которое можно затем «подмешать» (to mixin) к некоторому существующему элементу.

Основное отличие примесей от интерфейсов – законченность. Хотя примеси и могут иметь абстрактные члены, всё же на практике большинство полей, типов и методов полностью определены. Примеси впервые были использованы в языках Flavors и CLOS. Среди современных языков в чистом виде поддерживаются в Ruby, Scala и Python.

В языке Scala в роли такой функциональной единицы выступает трейт (Trait).

Трейт Simulator

Трейт Simulator является базовым модулем, к которому подмешиваются другие элементы, уточняя его поведение. Фактически, Simulator, – это каркас для логического процесса.



В трейте 2 члена являются абстрактными:

* type T – тип произвольного сериализуемого объекта, который представляет собой состояние модели
* startModelling: () => T – метод инициализации, который возвращает начальное состояние модели

Таким образом, в реализующем классе следует указать лишь тип для состояния и определить метод *startModelling*, который вернёт начальное состояние заданного типа.

**Поля**

* *private time: Float* – модельное время логического процесса
* *private state: T* – состояние логического процесса, заданное некоторым сериализуемым типом T

**Функции**

* *startModelling: () => T* – единственная функция, которая нуждается в реализации. Здесь выполняется код инициализации и возвращается начальное состояние перед запуском моделирования
* *getTime: () => Float* – возвращает текущее время
* *getState: () => T* – возвращает текущее состояние
* *commitEvent: TimeEvent => ()* – самая главная функция, суть которой –продвинуть время вперёд.
* *onReceive: PartialFunction[Any => ()]* – частично заданная функция, используемая для классификации принятых сообщений
* *sendMessage: (String, Message) => ()* – отправляет сообщение конкретному логическому процессу
* *sendMessageToAll: Message => ()* – отправляет сообщение всем логическим процессам
* *private setStateAndTime: (Float, T) => Unit* – закрытая для пользователя функция, изменяющая локальные время и состояние

**Важное замечание**: изменять состояние логического процесса допускается строго через функции *getState* и *commitEvent*. В этом случае оптимистический алгоритм сможет выполнить снимок текущего состояния и сохранить его в стеке состояний. Если пользовательский класс будет изменять состояние другим способом, то в будущем при возникновении отката могут возникнуть ошибки.

Реализация симулятора (первый прототип)

Трейт Communicator

Трейт *Communicator* представляет собой модуль для работы с акторской системой. Он содержит в себе экземпляр актора, который занимается посылкой/приёмом сообщений. Трейт примешивается к трейту Simulator:

**trait** Simulator[T <: *Serializable*] **extends** Communicator[T]



Трейт содержит одну абстрактную функцию – *onReceive* – которая реализуется в симуляторе, и один вложенный класс – Receiver – который наследует от актора. Ниже перечислены члены трейта:

**Поля**

* *private conf* – специальный объект, хранящий конфигурацию акторской системы
* *private systemname* – имя акторской системы (извлекается из объекта *conf*)
* *private actorAddresses* – адреса всех соседей (извлекается из объекта *conf*)
* *system* – объект, представляющий собой акторскую систему
* *actorname* – текущее имя актора
* *actornames* – имена акторов-соседей
* *actors* – ассоциативный массив, каждому имени актора ставящий в соответствие ссылку на актора

**Функции**

* o*nReceive: PartialFunction[Any => ()]* – абстрактная функция, классифицирующая принятые сообщения. Реализована в Симуляторе

Данный трейт в пользовательском коде не используется. За отправку сообщений отвечают описанные выше функции *sendMessage* и *sendMessageToAll*, а за приём – другие трейты, которые будут описаны в последующих разделах.

Трейт OptimisticSynchronizator

Данный трейт является ключевым звеном для организации PDES – он реализует оптимистический алгоритм синхронизации Time Warp. Любопытно отметить, что Симулятор можно оснастить таким мощным инструментом, всего лишь подмешав этот трейт:

**trait** Simulator[T <: *Serializable*] **extends** Communicator[T] with OptimisticSynchronizator[T]



**Поля**

* *private stateStack: Deque[TimeEvent -> Array[Byte]* – стек состояний – хранит все снимки состояний на случай отката
* *private msgStack: Deque[EventMessage -> String]* – стек сообщений – хранит все отправленные сообщения на случай отправки антисообщений при откате
* *private inputQueue: List[BaseMessage]* – буфер для входящих сообщений

Обратите внимание – для организации стека применяется двухвостая очередь (*Deque*), а не собственно стек (*Stack*). Вызвано это необходимостью доступа не только к началу коллекции, но и к концу – после вычисления GVT синхронизатор должен освободить память, удалив старые снимки состояний и сохранённые сообщения.

**Функции**

* *stopModelling: () => Statistics* – функция, вызываемая при окончании моделирования; пользователь может (более того, ему рекомендуется) переопределить функцию, не забыв вызвать ***super***. В данном трейте функция очищает все вышеперечисленные контейнеры и также вызывает ***super***.
* *sendMessage: (String, message) => ()* – функция отправки сообщений – сохраняет сообщение в стек. Симулятор, переопределяя эту функцию, должен сделать вызов ***super***.
* *commitEvent: TimeEvent => ()* – как было отмечено в разделе «Трейт Simulator», функция продвигает время вперёд. Здесь же выполняется очень важное действие – выполняется снимок состояния, и затем этот снимок помещается в стек. Симулятор, переопределяя эту функцию, должен сделать вызов ***super***.
* *peekMessage: () => Option[EventMessage]* – возвращает текущее сообщение из входного буфера, но не удаляет его
* *popMessage: () => Option[EventMessage]* – возвращает текущее сообщение из входного буфера. Само сообщение из буфера удаляется
* *private timeIsLessThanMessage: (Double, BaseMessage) => Boolean* – небольшая служебная функция, проверяющая, что заданное время меньше, чем временнáя метка сообщения: причём знак строгий для антисообщений, и нестрогий – для «позитивных» сообщений. Актуальность этой функции см. в примечании к Алгоритму#3 (раздел «Общий алгоритм обработчика сообщений»).
* *private calculateGVT: () => Boolean* – вычисляет текущую оценку GVT
* private *rollback: BaseMessage => ()* – выполняет откат на предыдущее согласованное состояние
* *private handleMessage: BaseMessage => ()* – функция, вызываемая актором для обработки поступивших сообщений. Основной код функции представлен в Алгоритме#3 в разделе «Общий алгоритм обработчика сообщений»
* *private snapshot: TimeEvent => ()* – выполняет снимок состояния логического процесса
* *private resetBuffers: () => ()* – очищает стек состояний, стек сообщений буфер входящих сообщений

Данный трейт добавляет одну полезную пользовательскую функцию – *stopModelling*. Переопределим её в нашем первом прототипе логического процесса:

**public class** AbstractModel **extends** Simulator<State> {

**private** **transient** *Cancellable* \_timer;

@Override

**public** State *startModelling*() {

...

}

@Override

**public** Statistics *stopModelling*() {

\_timer.cancel();

**return** **super**.stopModelling();

}

}

Трейт ModelObservable

Данный трейт является реализацией концепции *информационных процедур*, принятых в области имитационного моделирования для организации сбора статистики.

Определение. Информационные процедуры – программные компоненты, накладываемые на модель с целью сбора статистики в ходе имитационного прогона

Трейт может быть примешан как к Симулятору, так и к Синхронизатору. В нашем примере он примешан к последнему:

**trait** OptimisticSynchronizator[T <: *Serializable*] **extends** ModelObservable

Трейт вводит новый тип – *Statistics*. Он представляет собой ассоциативный массив, каждой категории ставящий в соответствие некоторое число. Всего на этот случай было выделено 18 категорий:

* EVENTS\_HANDLED – число обработанных событий
* RECEIVED\_MESSAGES – число полученных сообщений
* RECEIVED\_EVENT\_MESSAGES – число полученных информационных сообщений
* RECEIVED\_ANTI\_MESSAGES – число полученных антисообщений
* SENT\_MESSAGES – число отправленных сообщений
* SENT\_EVENT\_MESSAGES – число отправленных информационных сообщений
* SENT\_ANTI\_MESSAGES – число отправленных антисообщений
* ROLLBACKS – число откатов
* ROLLBACKS\_MAXDEPTH – максимальная глубина отката (ед. времени)
* ROLLBACKS\_DEPTH\_1 – число откатов глубины 1
* ROLLBACKS\_DEPTH\_2 – число откатов глубины 2
* ROLLBACKS\_DEPTH\_3 – число откатов глубины 3
* ROLLBACKS\_DEPTH\_4 – число откатов глубины 4
* ROLLBACKS\_DEPTH\_5 – число откатов глубины 5
* ROLLBACKS\_DEPTH\_6 – число откатов глубины 6
* ROLLBACKS\_DEPTH\_7 – число откатов глубины 7
* ROLLBACKS\_DEPTH\_MORE – число откатов глубины 8 и более
* MAX\_TIME\_WINDOW – максимальный размер временнóго окна – разницы модельного времени до и после отката



**Поля**

* *private statistics: Statistics* – текущая статистика имитационного прогона

**Функции**

* *stopModelling: () => Statistics* – функция, описанная ранее в предыдущем разделе. Здесь производится очистка массива *statistics*. Трейт *OptimisticSynchronizator* обязан вызвать ***super*** при переопределении метода.
* *statMessageSent: BaseMessage => ()* – функция обработки отправляемых сообщений
* *statMessageReceived: BaseMessage => ()* – функция обработки принятых сообщений
* *statRollback: (Int, Double) => ()* – функция обработки откатов (на вход принимает глубину отката и размер временнóго окна)
* *statEventHandled: () => ()* – функция обработки событий

Трейт не предназначен для использования в пользовательском коде. Для получения статистики следует послать логическому процессу специальное сообщение *StopMessage* (см. раздел «Структуры данных, используемые в системе»), на что актор вызовет метод *stopModelling*, который вернёт собранную статистику.

Трейт Analyzer

Трейт Analyzer является, пожалуй, ключевым элементом всей исследовательской работы. Он добавляет «интеллектуальность» алгоритмам синхронизации: именно здесь реализованы алгоритмы, основанные на знаниях. Очевидно, Analyzer примешивается к трейту OptimisticSynchronizator:

**trait** OptimisticSynchronizator[T <: *Serializable*] **extends** ModelObservable **with** Analyser[T]



Данный трейт добавляет ещё 3 абстрактных функции, которые пользователь должен реализовать:

* *suspendModelling*
* *simulateStep*
* *getActorName*

Таким образом, итоговый симулятор обязан реализовать 4 функции (перечисленные выше и *startModelling*). Подробный пример будет представлен в следующем параграфе.

**Поля**

* *private lockingEvent: Option[AgentEvent -> AgentEvent]* – является частью Алгоритма#2. Содержит данные о блокирующем событии и разблокирующем событии

**Функции**

* *suspendModelling: Boolean => ()* – абстрактная функция, которая должна быть реализована в реальном симуляторе. Вызывается Алгоримом#2, если требуется заблокировать/разблокировать логический процесс
* *simulateStep: TimeEvent => Array[TimeEvent]* – абстрактная функция, которая должна быть реализована в реальном симуляторе. Вызывается Алгоритмом#3 для того, чтобы выполнить «псевдообработку» события. Функция возвращает список событий-потомков, которые впоследствии будут проверяться на каузальную зависимость. В главе 6 будет показано, как реализовать эту функцию, используя знания о модели
* *getActorName: AgentEvent => String* – абстрактная функция, которая должна быть реализована в реальном симуляторе. Вызывается Алгоритмом#2 для того, чтобы конкретному событию сопоставить имя актора, которому следует послать сообщение TimeRequest
* *commitEvent: TimeEvent => Unit* – функция, которая дважды переопределяется в Синхронизаторе и Симуляторе. В данном трейте функция является частью Алгоритма#2: она обращается к подсистеме знаний, чтобы проверить, требуется ли заблокировать логический процесс. Все трейты, переопределяя данную функцию, обязаны вызвать ***super***.
* *private handleLockRequest: LockRequest => ()* – функция-обработчик сообщения LockRequest. Отвечает отправителю сообщением LockResponse, если процесс заблокирован (дабы избежать взаимных блокировок)
* *private handleLockResponse: () => ()* – немедленно снимает блокировку логического процесса
* *private checkUpMessage: EventMessage => ()* – часть Алгоритма#2: проверяет, является ли сообщение разблокирующим (для возобновления работы логического процесса)
* *private resumeByAntimessage: Antimessage => ()* – снимает блокировку логического процесса ввиду того, что при откате процесс отправляет антисообщение, которое может привести к бесконечному ожиданию (см. Алгоритм#2 раздел «Проблема бесконечного ожидания»)
* *private isOK: TimeEvent => Boolean* – точка входа в Алгоритм#3. Функция подробно описана в Алгоритме#3 (раздел «Общий алгоритм обработчика сообщений»).
* *private suspend: (AgentEvent, AgentEvent) => ()* – часть Алгоритма#2: выполняет блокировку логического процесса, вызванную некоторым блокирующим событием, ожидающего соответствующего разблокирующего события
* *private resume: () => ()* – снимает блокировку логического процесса
* *private correlate: (AgentEvent, AgentEvent) => Boolean* – проверяет, существует ли гипотетическая зависимость между событиями (согласно *необходимому условию каузальной зависимости*). Реализация предложена в Алгоритме#3 (раздел «Общий алгоритм обработчика сообщений»)
* *private isLocal: AgentEvent => Boolean* – проверяет, является ли событие локальным (т.е. должно ли оно выполняться на локальном процессе)
* *private isSafe: TimeEvent => Boolean* – проверяет, является ли событие безопасным (Алгоритм#3). Подробное описание и теоретическая оценка сложности данной функции представлены в главе 4, п. Алгоритм#3.
* *private runPseudoEvent: TimeEvent => Boolean* – запускает «псевдообработку» события, выполняет фильтрацию полученных событий-потомков по времени, и рекурсивно вызывает функцию *isOK* для каждого такого события. Подробнее см. Алгоритм#3 (раздел «Общий алгоритм обработчика сообщений»)

Реализация простейшего логического процесса

В данном параграфе будет предложена реализация простейшего логического процесса на базе платформы, описанной выше. Библиотека позволяет реализовать симулятор на любых Java-based языках, таких как Java, Groovy, Scala и др. В нашем случае возьмём, для примера, язык Java.

Сначала начнём с простейшего прототипа, не учитывающего трейт Analyzer. Позже мы реализуем недостающие функции для второго прототипа. Итак, для создания логического процесса следует:

* Создать класс, представляющий собой состояние логического процесса (он и будет типом T)
* Создать класс, реализующий трейт *Simulator*[[1]](#footnote-1), у которого параметром типа выступает T
* Реализовать метод *startModelling*, возвращающего начальное состояние типа T
* Учесть, что любые изменения состояния следует проводить через функцию *getState*, поскольку только она гарантирует, что откаты будут выполняться верно

Состояние логического процесса

Ниже представлена простейшая реализация состояния логического процесса (код упрощён для наглядности)

**public class** State **implements** Serializable {

**public** Map<String, String> remoteAgents;

**public** Map<String, Agent> agents;

**public** State(Map<String, Agent> agents, Map<String, String> remoteAgents) {

**this**.agents = **new** ConcurrentHashMap<>(agents);

**this**.remoteAgents = **new** ConcurrentHashMap<>(remoteAgents);

}

}

Как видно, состояние хранит два ассоциативных массива: первый содержит перечень удалённых агентов (каждому имени агента ставится в соответствие имя удалённого узла, на котором он выполянется), второй – перечень локальных агентов (каждому имени агента ставится в соответствие ссылка на агента).

Реализация симулятора: первый прототип

Реализуем Simulator и определим в нём базовую функцию – *startModelling*, которая запускается при получении сообщения *StartMessage*.

**public class** AbstractModel **extends** JavaSimulator<State> {

**private transient** Cancellable \_timer;

**protected transient** Map<String, Agent> \_agents = new TreeMap<>();

**protected transient** Map<String, String> \_remoteAgents = new TreeMap<>();

***@Override***

**public** State *startModelling*() {

// запускаем таймер

**final** Simulator<State> self = **this**;

\_timer = system().scheduler().schedule(..., **new** Runnable() {

***@Override***

**public void** *run*() {

**synchronized** (self) {

// поиск агента с минимальной временной меткой

Agent cur\_agent = **null**;

**for** (Agent agent : *getState*().agents.*values*()) {

...

}

Float t1 = cur\_agent != **null** ? cur\_agent.*getCurrentTimestamp*() : **null**;

// поглядим, есть ли что-то во входящей очереди

**try** {

t2 = *peekMessage*().*get*().t();

} **catch** (Exception ex) {...}

// выборка события с меньшей временной меткой

TimeEvent event = ... //Выбор минимума среди t1 и t2

// обработка события

String remoteActorname = *getState*().remoteAgents.*get* (event.event().patiens());

**if** (remoteActorname != **null**)

*sendMessage*(remoteActorname, **new** EventMessage(actorname(), event));

**else** *getState*().agents.*get*(event.event().patiens()). *addEvents*(*runAgent*(event));

*commitEvent*(event);

}

}

}, ...);

// сброс агентов

**for** (Agent agent : \_agents.*values*())

agent.*flush*();

// создаём новый экземпляр State

**return new** State(\_agents, \_remoteAgents);

}

**private** Collection<TimeEvent> *runAgent*(TimeEvent event) {

Collection<TimeEvent> result = Collections.emptyList();

**if** (getState().agents.containsKey(event.event().patiens()))

**try** {

Agent r = *getState*().agents.*get*(event.event().patiens());

Object newEvents = r.*getClass*().*getMethod*(event.event().predicate(), TimeEvent.**class**).*invoke*(recipient, event);

result = (Collection<TimeEvent>) newEvents;

} **catch** (Exception ex) {...}

**return** result;

}

}

В инициализирующей функции *startModelling* запускается таймер, который периодически извлекает события, обрабатывает их и в конце вызывает функцию *commitEvent*. Это продвигает время до отметки обработанного события. События выбираются в хронологическом порядке либо из агентов, либо из входного буфера.

Обратите внимание на 2 важных обстоятельства: во-первых, метод **единожды** создаёт состояние, которые возвращается на выходе. Во-вторых, состояние модели изменяется **строго** через функцию *getState*. Это гарантирует, что откаты будут выполняться корректно.

Вспомогательный метод *runAgent* выполняет предикат события внутри агента через Java Reflection API. На выходе данный метод возвращает список новых событий, сгенерированных агентом.

Реализация симулятора: второй прототип

Предложенная выше реализация является полной лишь до тех пор, пока не будет подмешан трейт *Analyzer*, содержащий в себе алгоритмы синхронизации, основанные на знаниях о модели. Как известно, в таком случае потребуется реализовать ещё 3 метода:

* *suspendModelling*
* *simulateStep*
* *getActorName*

Ниже представлено дополнение к существующей реализации первого прототипа:

**public class** AbstractModel **extends** JavaSimulator<State> {

**private volatile boolean** \_locked = **false**; //блокировка

...

***@Override***

**public** State *startModelling*() {

// запускаем таймер

**final** Simulator<State> self = **this**;

\_timer = system().scheduler().schedule(..., **new** Runnable() {

***@Override***

**public void** *run*() {

**synchronized** (self) {

// поиск агента с минимальной временной меткой

...

// поглядим, есть ли что-то во входящей очереди

...

// посмотрим, не нужна ли блокировка

**if** (t1 < *getTime*() || t2 < *getTime*()) {

**if** (\_locked) logger().debug("Событие из прошлого!");

} **else if** (\_locked) **return**;

// выборка события с меньшей временной меткой

...

// обработка события

...

}

}

}, ...);

...

}

**@Override**

**synchronized public void** *suspendModelling*(**boolean** suspend) {

\_locked = suspend;

}

**@Override**

**public** TimeEvent[] *simulateStep*(TimeEvent e) {

Collection<TimeEvent> newEvents = *runAgent*(e);

**return** newEvents.*toArray*(**new** TimeEvent[newEvents.*size*()]);

}

**@Override**

**public** String *getActorName*(AgentEvent e) {

**if** (\_agents.containsKey(e.patiens())) **return** actorname();

**if** (\_remoteAgents.containsKey(e.patiens())) **return** \_remoteAgents.*get*(e.patiens());

**return null**;

}

**@Override**

**public** Statistics *stopModelling*() {

\_timer.*cancel*();

**return** **super**.*stopModelling*();

}

**private** Collection<TimeEvent> *runAgent*(TimeEvent event) {

...

}

}

Среди полей в классе добавилось лишь одно: \_*locked*. Оно показывает, заблокирован ли логический процесс[[2]](#footnote-2). В таймере перед выполнением события проверяется значение этой переменной. Если процесс заблокирован, то обработчик таймера прерывается. В реализованном методе *suspendModelling* переменная \_*locked* изменяет своё значение.

Обратите внимание: значение \_*locker* игнорируется, если обрабатывается сообщение «из прошлого». Это позволяет сделать интеграцию Алгоритма#2 и Алгоритма#3 более эффективной.

Реализованный метод *simulateStep* отвечает за «псевдообработку» события. Он просто вызывает существующий метод *runAgent*, который возвращает список событий-потомков.

Реализованный метод *getActorName* просматривает списки агентов \_*agents* и \_*remoteAgents*. Если пациенс события (т.е. имя агента) присутствует в локальном списке – возвращается имя текущего актора. Если же он имеется в списке удалённых акторов – возвращается имя удалённого актора.

Ну и наконец, метод *stopModelling* переопределяет существующий метод: он останавливает таймер и пробрасывает вызов через ***super***.

Выводы

На этом реализация простейшего логического процесса окончена. Отметим, что она является далеко не самой эффективной, но вполне пригодной для построения достаточно сложных моделей. Также предложенная реализация легко масштабируется и является потокобезопасной. Тестовая модель, о которой будет рассказано позже в главе 7, была построена на базе подобного симулятора.

Подведём краткий итог. Для создания логического процесса с помощью предложенной платформы требуется:

* Создать класс состояния
* Реализовать трейт *Simulator*
* Реализовать 4 метода, которые требует *Simulator*
* При необходимости переопределить др. методы (напр, *stopModelling*)
* По окончанию обработки каждого события вызывать *commitEvent*
* Учесть, что состояние меняется только через вызов функции *getState*

1. Способы представления знаний о модели. Разработка DSL
2. Эксперименты. Практические результаты

Заключение

Библиографический список

Глоссарий

Приложение

1. Необходимо учесть одно важное обстоятельство: кроме трейта Simulator библиотека предлагает также абстрактный класс *JavaSimulator*[T <: Serializable], который расширяет Simulator[T]. Это связано с тем, что Java-коду чуждо понятие трейта, и прямая его реализация может сопровождаться некоторыми затруднениями. [↑](#footnote-ref-1)
2. В Java существуют более эффективные методы блокировки: *wait*, ReentrantLock, Semaphore и др. В нашем примере для простоты используется простая булевская переменная [↑](#footnote-ref-2)