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 5](#_Toc359362744)

[Глава 1. Преимущества агентного имитационного моделирования 7](#_Toc359362745)

[Имитационное моделирование 7](#_Toc359362746)

[Мультиагентные системы 9](#_Toc359362747)

[Агентное моделирование 11](#_Toc359362748)

[Глава 2. Постановка задачи. Проблема синхронизации логических процессов 13](#_Toc359362749)

[Распределённое имитационное моделирование 13](#_Toc359362750)

[Консервативные алгоритмы синхронизации 17](#_Toc359362751)

[Оптимистические алгоритмы синхронизации 20](#_Toc359362752)

[Сравнение классов алгоритмов 23](#_Toc359362753)

[Глава 3. Оптимизация алгоритмов синхронизации логических процессов 26](#_Toc359362754)

[Модификации консервативных алгоритмов 26](#_Toc359362755)

[Модификации оптимистических алгоритмов 30](#_Toc359362756)

[Глава 4. Алгоритмы синхронизации агентов, основанные на знаниях о модели 42](#_Toc359362757)

[Ограничения и сфера применимости алгоритмов 42](#_Toc359362758)

[Природа откатов 44](#_Toc359362759)

[Алгоритм#1 46](#_Toc359362760)

[Алгоритм#2 47](#_Toc359362761)

[Алгоритм#3 50](#_Toc359362762)

[Глава 5. Разработка платформы. Архитектура, технологии и инструментальные средства 64](#_Toc359362763)

[Модель акторов 64](#_Toc359362764)

[Элементы языка Scala 66](#_Toc359362765)

[Структуры данных, используемые в системе 68](#_Toc359362766)

[Многослойная архитектура абстрактного симулятора 72](#_Toc359362767)

[Реализация простейшего логического процесса 82](#_Toc359362768)

[Глава 6. Способы представления знаний о модели 89](#_Toc359362769)

[Онтологический подход к представлению знаний о модели 89](#_Toc359362770)

[Предметная онтология и онтология задачи 91](#_Toc359362771)

[Пример онтологии тестовой модели 93](#_Toc359362772)

[Теоретическая оценка сложности с учётом знаний о модели 94](#_Toc359362773)

[Глава 7. Эксперименты. Практические результаты 97](#_Toc359362774)

[Общее описание модели 97](#_Toc359362775)

[Формализация агентной модели 98](#_Toc359362776)

[Результаты экспериментов 103](#_Toc359362777)

[Заключение 108](#_Toc359362778)

[Библиографический список 109](#_Toc359362779)

[Глоссарий 110](#_Toc359362780)

[Приложение 112](#_Toc359362781)

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*, *feedback-cycles*, *conditional events*, …). 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. 1).



Рис. . Схема последовательного симулятора

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 (рис. 2), и далее этот алгоритм зацикливается.



Рис. . Генерация новых событий в SDES

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



Рис. . Двухвостая очередь, используемая для списка событий

На этом, пожалуй, 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 (термин впервые введён Fudjimoto в 1990-м г.)

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. 4.



Рис. . Пример нарушения локальной каузальности

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. 5.



Рис. . Схема работы консервативного алгоритма

Из рисунка видно, что логический процесс не будет выполнять событие с меткой ***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. 6.



Рис. . Пример возникновения тупика

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, которое (семантически) несёт в себе данные только о временнóю метке[[1]](#footnote-1).

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. 7.



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:

*,* (1)

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. 8.



Рис. . Схема работы оптимистического алгоритма

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 изображён ниже (рис. 9).



Рис. . Сообщение, отправленное перед процедурой отката

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 каскадных 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 таблице 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 приёма являются Nicol и Rilay (1988г). Данный механизм является одним из самых эффективных, поскольку он использует знания о модели.

*Lookahead* даёт возможность предсказать, что случится с процессом моделирования. По факту, числовое значение lookahead показывает минимальный инкремент временной метки для обработки какого-либо события. Если логический процесс требует как минимум ***x*** ед. модельного времени для обработки следующего события, то говорят, что процесс имеет ***lookahead = x***. Тем самым он гарантирует, что он не сгенерирует сообщения с временнóй меткой, меньшей, чем ***cur\_time + x***. Таким образом, *lookahead* уменьшает требуемое число нулевых сообщений и вероятность возникновения тупика.

Сами Nicol и Relay предложили идею предварительного оценивания *lookahead* для событий и логических процессов. Само значение *lookahead* авторы называли *service time* (задержка обслуживания).

Carrier-Null Message

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

Общая идея заключается в том, что в каждое нулевое сообщение помещается вспомогательная информация, содержащая данные о текущих оценках *lookahead* и пройденных маршрутах для каждого типа сообщения. При пересылке сообщения логический процесс всякий раз уточняет текущие данные собственными оценками.

Следует отметить, что алгоритм работает только в так называемых *feed-backward networks* (сетях с обратными циклами). Это требуется для циркуляции оценок *lookahead* внутри графа логических процессов. Позже алгоритм был обобщён для сетей с мульти-иерархичным контуром управления (*LP network with multi-hierarchical feedback loops*).

Control flow graph

Cota и Sargent в 1990г. предложили фреймворк для автоматического вычисления *lookahead*. Для описания модели используется граф процессов управления (*control flow graph*), который описывает процессы и переходы состояний логический процессов. Далее при трансляции модели *lookahead* вычислялся на базе описанного графа.

За исходные данные берётся информация о так называемых *feedback loops* (контурах управления с обратной связью). При обнаружении таких циклов транслятор находит минимальные задержки времени по всем узлам цикла и вычисляет оценку *lookahead*.

Позже Lin, Lazowska и Baer в 1991г. опубликовали статью [], где был предложен класс систем, для которых невозможно выделить контуры обратной связи. Таким образом, они показали, что идея *Control flow graph* и *Carrier-null messages* будет эффективна лишь на весьма ограниченном наборе задач.

Deadlock detection and recovery algorithm

Авторами данного алгоритма выступают основатели PDES-синхронизации Chandy и Misra [].

Алгоритм синхронизации вместо того, чтобы избегать тупиков (*deadlocks*), наоборот, выполняет моделирование вплоть до возникновения тупика. Таким образом, для данного подхода требуется:

* механизмы обнаружения тупика;
* механизмы устранения тупика.

Для обнаружения тупика применяются стандартные методы, принятые для распределённых программ в целом. Для разрешения тупиков используется та идея, что сообщение с наименьшей временнóй меткой в системе всегда является безопасным для выполнения. Подробнее детали алгоритма изложены в [].

Основное преимущество предложенного подхода – полностью исключается потребность в нулевых сообщениях (правда, появляются дополнительные служебные сообщения, но их доля ничто мала по сравнению с нулевыми сообщениями в стандартных CMB-протоколах). Другое преимущество – отсутствие сообщений с нулевым инкрементом времени. В результате, эффективность вычислений резко возрастает, поскольку каждое сообщение обязательно продвигает модельное время.

Главным недостатком данного алгоритма авторы считают наличие моделей, склонных к тупикам (*prone-to-deadlock-models*). Типологическими особенностями таких моделей является:

* большое число связей между логическими процессами;
* относительно небольшое число передаваемых сообщений.

В таких моделях тупики могут возникать настолько часто, что их обнаружение и устранение может стать узким местом и стать причиной падения производительности.

Bounded lag simulation protocol

Некоторые исследователи предложили идею внесения специальных интервалов (моделируемого времени) в работу логических процессов для определения безопасных событий. Они предлагают понятие *дистанции* (*distance*), которое семантически схоже с концепцией *lookahead*. Под дистанцией понимают нижнюю границу инкремента моделируемого времени для всех необработанных сообщений. Другими словами, это минимальное количество модельного времени, которое потребуется событию одного логического процесса для того, чтобы повлиять на другое.

Lubachevsky в 1989г. придумал *Bounded lag simulation protocol* []. Иное название данного подхода – алгоритм консервативного временного окна (*Conservative time window algorithm*). Алгоритм предполагает, что если два события должны быть обработаны одновременно, то разница между их временными метками должны быть ограничена сверху некоторым числом (которое и считается размером временнóго окна).

Проблема как алгоритма Lubachevsky, так и всех его аналогов, – определение размера временнóго окна. По мнению специалистов, данная проблема является серьёзным ограничением, поскольку эта величина зависит от конкретной модели[[2]](#footnote-2), а значит, не существует универсального критерия её нахождения.

Conditional Events algorithm

Алгоритм условных событий был разработан Chandy и Sheirman в 1996г []. Общая концепция достаточно близка к той, что будет предложена в главе 4, поскольку авторы также используют внешние знания о модели.

Суть подхода заключается в том, что алгоритм каждый шаг выполняет классификацию необработанных событий на условные (*conditional events*) и безусловные (*definite events*). Безусловные события будут точно выполнены в ходе моделирования, они не могут быть отменены или изменены обработкой других событий. К ним относят все уже существующие в системе события, а также те гипотетические события, чьи вероятности обработки равны единице. Условные события наоборот, будут обработаны только лишь при некоторых обстоятельствах.

Итак, основная мысль алгоритма – безусловные события всегда безопасны для обработки. Специальные программные компоненты выполняют классификацию событий на основе некоторой вспомогательной информации о модели. Затем безусловные события подлежат обработке.

Dag Consistent Parallel Simulation

Lim, Low и Turner в 1997-м г. предложили весьма оригинальный подход, названный *Dag Consistent Algorithm* []. Смысл заключается в том, что весь процесс моделирования делится на циклы, внутри которых логика продвижения времени делится между всеми логическими процессами на так называемые Cilk-процедуры (фрагмент вычисления логического процесса, который может быть отдан другому процессу).

На этапе стартового цикла алгоритм случайным образом (или на базе некоторых эвристик, основанных на топологии сети логических процессов) создаёт иерархического дерево Cilk-процедур. Затем специальная компонента (*The divide and conquer procedure*, – от англ, *процедура “разделяй и властвуй”*) отправляет глобальное время от корня дерева всем свои потомкам (т.е. всем логическим процессам). Под глобальным временем понимают наименьшую временнýю метку всех необработанных событий.

Далее все логические процессы вычисляют свою оценку безопасного времени (*safe time*) – значения времени, вплоть до которого обработка всех событий безопасна. После обработки безопасных событий логический процесс возвращает новую оценку безопасного времени (равную минимуму временных меток из локального календаря и всех входящих буферов). Все эти значения возвращаются обратно от вершин дерева к корню, и затем *The divide and conquer procedure* агрегирует собранные данные и переходит к следующему циклу обработки.

Главное преимущество данного подхода – полный отказ от нулевых сообщений. Это, очевидно, снижает издержки и уменьшает нагрузку на сеть. Основной недостаток – восприимчивость к тесносвязанным моделям. В этом случае циклы получаются достаточно маленькими, что неблагоприятно сказывается на производительности.

Прочие алгоритмы

В литературе известны и другие решения по оптимизации консервативных алгоритмов. Так, например, Kumar в 1989г. предложил весьма эффективный алгоритм синхронизации для ациклических сетей []. Nevison в 1996г. предложил протокол координации модельного времени для сетей с многочисленными циклами []. Vries в 1999г. предложил сети с компонентами обратной связи (*networks with feedforward & feedback components*) [].

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*).

При отложенной отмене вместо немедленной отправки антисообщений во время отката, логический процесс отправляет их в выходной буфер, завершает процесс отката и возобновляет моделирование с предыдущего согласованного состояния. Антисообщения отправляются из буфера только в том случае, если его соответствующее «позитивное» сообщение не было сгенерировано вновь.

По словам Fujimoto, предложенный алгоритм использует *неявный lookahead* (*implicit lookahead*), который в отличие от консервативных алгоритмов не следует задавать заранее.

Подробное сравнение схемы принудительной и отложенной отмены представлено в []. Несмотря на то, что отложенная отмена значительно снижает нагрузку на сеть, она может увеличить время, требуемое на глубокие каскадные откаты. Связано это с тем, что логический процесс не сразу отправляет антисообщения, а лишь в случае необходимости после перезапуска моделирования. Автор утверждает, что можно искусственно придумать ситуации, когда та или иная стратегия окажется более эффективной.

Adaptive Cancellation Algorithm

Автор алгоритма – Wilsey. Общая идея заключается в том, чтобы комбинировать принудительную и отложенную отмену так, чтобы добиться максимальной эффективности. Отложенная отмена в целом более эффективна, но при больших каскадных откатах она может снижать производительность из-за задержек времени. Поэтому алгоритм предлагает каким-либо образом оценивать ситуацию и принимать решение, какая схема отмены будет наиболее предпочтительной.

Jump Forward Algorithm

Метод также носит название «Алгоритм отложенного отката» (*Lazy reevaluation* или *Lazy rollback algorithm*). Автор – Fujimoto (1990г.) []. Фундаментальная идея данного алгоритма была взята за основу предлагаемого в данной работе Алгоритма#3 (см. главу 4). В отличие от нашего алгоритма, Jump Forward не использует знания о модели.

Суть алгоритма заключается в том, что при получении сообщения «из прошлого» логический процесс выполняет «виртуальный» откат, после этого возобновляет работу с прежнего согласованного состояния и сравнивает векторы состояния до и после отката. Если эти векторы равны, то нет необходимости делать откат всех промежуточных состояний, и алгоритм попросту «перепрыгивает» (*jumps forward*) через них.

Сам Fujimoto предупреждает о необходимости ведения специального журнала регистрации векторов состояний модели, что делает данный подход более требовательным к ресурсам.

Wolf Calls Algorithm

Madissetti, Warland и Messerschmiht в 1988г. [] предложили алгоритм «Волчьего воя». Цель данного подхода – ограничить распространение некорректных вычислений в случае нарушения локальной каузальности одного из логических процессов.

Идея очень проста – как только логический процесс получает struggling-сообщение, он тут же отсылает всем своим соседям служебное сообщение с просьбой заблокировать все вычисления на определённой временнóй метке. Тем самым, логический процесс оберегает своих соседей от глубоких каскадных откатов.

Получив служебное сообщение, процесс блокируется и дублирует сообщение всем остальным соседям. Таким образом, мы получили одну из схем «сдерживания оптимизма».

Существуют различные модификации данного алгоритма. Связаны они, прежде всего, с тем, что служебное сообщение «замораживает» все вычисления, в т.ч. корректные. Авторы не отрицают возможность применения дополнительных знаний, но предупреждают, что далеко не всегда они могут быть доступны в конкретных моделях.

Moving Time Windows

Алгоритм, предложенный авторами Sokol, Brisco и Wailend в 1988г., до сих пор остаётся самой популярной идеей борьбы с «чрезмерным оптимизмом» []. Как и в случае консервативных алгоритмов, вводится понятие временнóго окна – некоторого промежутка модельного времени, традиционно обозначаемого через W.

Идея крайне проста – логический процесс имеет право обрабатывать события лишь из промежутка [GVT; GVT+W]. Реже применяется промежуток [CurTime; CurTime+W]. Как и в случае консервативного Time Windows, ключевой проблемой алгоритма является определение числа W. Точных оценок не существует, эмпирические оценки приведены в работах [].

Существуют научные работы, показывающие, как в одних случаях MTW работает превосходно, в других – ничуть не превосходит обычный Time Warp. Немало работ посвящено подбору идеальной величины параметра ***W***. Ниже будут приведены два наиболее популярных подхода: один из них использует оценки на базе теории вероятностей, другой – генетические алгоритмы.

Filtered Rollback Algorithm

Идея, предложенная Swartz и Weitson в 1989г. [], фактически является объединением оптимистического *Moving Time Windows* и консервативного алгоритма Лубачевского *Bounded Lag*.

*Filtered rollback* содержит настраиваемые параметры, которые в одном пограничном значении делают алгоритм эквивалентом MTW, а в другом пограничном значении – эквивалентом *Bounded Lag*.

Direct Cancellation Algorithm

Механизм *Прямой Отмены* впервые ввёл Fujimoto (1991г.)[] для архитектур с общей памятью. Суть данного подхода заключается в следующем: для каждого обрабатываемого события в логическом процессе предусмотрены указатели на все те состояния, куда может произойти откат. При получении сообщения «из прошлого» логический процесс просто переходит по указателю и, тем самым, восстанавливает согласованное состояние.

Нетрудно показать, что в плане производительности данный подход на порядок превосходит идею использования антисообщений. Однако, недостаток данного подхода весьма очевиден – поддержка только SMP-архитектур, которые, как известно, прекрасно работают при небольшом количестве процессоров, но начинают стремительно терять в масштабируемости по мере их роста.

Space-Time Simulation

Chandy и Sheirman в 1992г. [] предложили довольно экзотическую идею, получившую название *пространственно-временное моделирование* (*Space-Time Simulation*). Суть такого подхода заключается в том, что весь процесс моделирования представляет собой заполнение двумерной прямоугольной сетки соответствующими значениями. По оси абсцисс откладывается моделируемое время, по оси ординат – компоненты вектора состояния модели (рис. 10).

Идея алгоритма – разделить полученную двумерную область на непересекающиеся фрагменты и назначить каждому логическому процессу задачу заполнить один из таких фрагментов.



Рис. 10. Двумерная сетка для Space-Time Simulation

Таким образом, каждый логический процесс отвечает лишь за свой фрагмент плоскости. Единственная проблема – выполнение граничных условий. Условия одного фрагмента должны полностью совпадать с условиями соседнего фрагмента вдоль всей линии границы. Если это не так, выполняется перевычисление. Если при перевычислении снова найдена несогласованность, процесс повторяется.

Fujimoto в своей статье [] заметил, что такой способ моделирования концептуально эквивалентен алгоритму Time Warp со стратегией отложенной отмены (а значит, имеет все те же преимущества и недостатки).

Filtering Algorithm

Prakash и Subramanian в 1992г. предложили *Filtering Algorithm* (*Алгоритм фильтрации* []). Он немного похож на описанный выше *Wolf Calls*, поскольку ставит основной целью сдерживание глубоких каскадных откатов. Однако здесь алгоритм не «замораживает» вычисления, а заставляет логические процессы собирать информацию во время имитационного прогона: количество откатов, количество пересылаемых сообщений и т.д. Далее эта информация некоторым способом агрегируется и отсылается всем соседям.

Преимущества и недостатки подхода весьма очевидны: за выигрыш в производительности приходится платить дополнительными издержками на сбор, обработку и пересылку служебных данных.

Breathing Time Buckets

Предложенный автором Steinman в 1996г. [] 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. Вообще, данный алгоритм является вторым по популярности оптимистическим алгоритмом после Time Warp. Существует немало работ, посвящённых сравнению этих двух алгоритмов [], большинство из которых доказывают более высокую эффективность метода *Breathing Time Buckets*.

Алгоритм *Breathing Time Buckets* имеет главное преимущество: все откаты являются локальными, а значит, как следствие, справедливы следующие утверждения:

* не требуются антисообщения;
* не требуется память под отправленные сообщения;
* отсутствуют каскадные откаты.

Логические процессы в цикле продвигают время моделирования: обрабатывают события, обмениваются сообщениями и при необходимости выполняют откат. В каждом цикле обрабатываются все события в порядке неубывания временных меток.

Предположим, что с некоторого момента логический процесс начинает обрабатывать события, генерирующие другие события. Минимальная метка всех сгенерированных событий называется *Глобальным Временем* (*GST[[3]](#footnote-3), Global Simulation Time*). Поскольку все вычисления, проводимые от текущего времени до GST не могут предшествовать друг другу, то их можно запускать параллельно.

Каждый процесс вычисляет собственную нижнюю оценку GST, называемую локальным горизонтом событий (*LEH*, *Local Event Horizon*).

GST рассчитывается как минимум всех меток LEH. Стоит отметить, что локальные метки LEH ≥ GST для каждого логического процесса, поэтому некоторые события будут обработаны «оптимистично» (с забеганием вперёд).

Те события, которые были обработаны с нарушением локальной каузальности, могут быть отменены. При этом, очевидно, анти-сообщения не требуются, поскольку откаты будут локальными.

Выше была предложена идея основного алгоритма. Оптимизация алгоритма (Booth, 1993) осуществляется за счёт информирования логическими процессами друг друга о своей нижней оценке LEH. Получив такое информационное сообщение, логический процесс может понизить текущую оценку горизонта, чтобы избежать чрезмерного забегания вперёд, которое впоследствии приведёт к откатам.

Как мы видим, предложенный алгоритм также преследует политику сдерживания «чрезмерного оптимизма». В отличие от Moving Time Windows, допустимая величина «забегания» вперёд задаётся не константой, а оценкой минимума всех будущих событий.

Genetic Algorithm for Moving Time Windows

Jun Wang и Carl Tropper в 2009-м году разработали генетический алгоритм для вычисления оптимального размера временнóго окна W для алгоритма *Moving Time Windows*. Напомним, что дальше данного окна вычисления не производятся, поскольку увеличивается риск нарушения локальной каузальности.

В качестве функции представления генотипа (*mapping genotype function*) выбирается размер окна в двоичном представлении (бинарное разложение или код Грея). Таким образом, ***f(x)=ωx***, где ***ω*** – некоторая константа. Константа может различаться в зависимости от конкретного вычислительного узла (и таким образом, размер окна может быть разным для разных узлов).

Фитнесс-функция задаётся эмпирически. Для оценки авторы предлагают оценить вектор размеров окна, поместив окно в конечное состояние вычисления GVT и измерив приращения времени моделирования на конец следующего временного интервала GVT. Таким образом, функция определяется как:

где ***n*** – число логических процессов;

***e*** – число корректных (*commited*) сообщений (имеется в виду тех сообщений, которые успешно обработаны без откатов);

***∆t*** – оценка временного интервала между перерасчётом GVT.

Для правильной аппроксимации фитнесс-функции авторы рекомендуют вычислить не менее 10 перерасчётов GVT.

Для селекции были рекомендованы 2 метода: метод рулетки и метод вероятностной равномерной выборки (*Stochastic universal sampling*, *SUS*), а для кроссовера – метод одноточечного кроссовера.

В качестве мутации могут быть применены любые методы в зависимости от представления генов. После кроссовера и мутации потомки и их родители объединяются в общее поколение, сортируются по фитнесс-функции, и наиболее слабые особи удаляются из поколения.

Нетрудно заметить, что данный алгоритм относится к классу адаптивных (как *Adaptive Cancellation* и *Filtered Rollback*), поскольку он учитывает интенсивность входящего потока сообщений и каждый раз пересчитывает размер окна. К недостаткам можно отнести эвристическую оценку фитнесс-функции и значительное увеличение вычислительной сложности алгоритма в силу работы ГА.

Rollback Relaxation Algorithm

Wilsey, Palaniswamy и Aji в 1994-м году предложили метод, повышающий эффективность моделирования физических процессов. Общая идея заключается в том, что моделируемые процессы делятся на *события, зависящие от памяти* (*memoried events*) и *события, не зависящие от памяти* (*memoryless events*). Первые встречаются в тех процессах, выход которых представляет собой функцию от входа и от внутренних переменных. Вторые, соответственно, – функцию только от входа.

Схема алгоритма выглядит примерно следующим образом: все входные сообщения в буфере делятся на 2 подмножества: *memoried* и *memoryless*. Далее происходит сортировка по двум критериям: по временнóй метке события и переменным, на которые оно может повлиять. Эта информация позволит логическим процессам быстро определить входные значения всех переменных при откате, чтобы затем продолжить моделирование не с самого раннего события, а лишь с того, которое может вызвать парадоксы времени. Достигается это путём простого сравнения переменных.

Данный алгоритм является достаточно эффективным, т.к. он использует некоторые знания о модели – информацию о том, какие переменные могут быть изменены тем или иным событием, можно успешно применить для уменьшения откатов.

Probabilistic Time Window Adaptation

Немецкий исследователь Johannes Luthi в 1996-м предложил вероятностную модель для MTW-протокола. Автор в своей работе [] предлагает *вероятностную функцию принятия решения* (*Probabilistic Decision Function*, DF), а затем приводит несколько конкретных примеров её реализации. Сначала рассмотрим формальное определение DF:

Пусть Sclock обозначает процессорное время (от момента старта моделирования). Пусть H – множество возможных вариантов исхода моделирования (*simulation histories*), а E – множество всевозможных событий. За τ(e) обозначим временнýю метку события ***e***. Определим функцию ***DF*** следующим образом:

Функция должна монотонно убывать относительно временнóй метки:

,

и не должна вызывать тупиков:

Варьируя характер данной функции, можно предложить адаптивный протокол для гибкой настройки управления оптимизмом.

Теперь рассмотрим один из примеров реализации конкретного алгоритма. Идея заключается в том, чтобы разницу модельного времени между успешно обработанными сообщениями логического процесса положить в качестве случайной величины с некоторым распределением. Предполагается, что набор таких значений является НОРСВ. Автор предлагает несколько семейств распределений (как параметрических, так и непараметрических) и техники оценивания параметров с применением экспоненциального сглаживания (в частности, для нормального распределения).

Пусть ***ti*** – время ***i***-го полученного сообщения (не вызвавшего отката) и пусть ***∆e = te- tn*** – разница между временнóй меткой ***te*** следующего события ***e*** и временнóй меткой ***tn*** последнего полученного сообщения ***mn***. Предположим, была получена следующая последовательность функций (рис. 11):

Эти функции оценивают вероятность того, что событие ***e*** с инкрементом времени ***∆e*** относительно полученного сообщения ***mn*** вызовет откат с приходом сообщения ***mn+1*** с меткой ***tn+1***. Как только величина достигнет некоторого порогового значения, логический процесс блокируется до момента прихода сообщения ***mn+1***.

Отметим, что блокировка служит потенциальной угрозой возникновения тупиков (*deadlocks*), поскольку процесс не гарантирует, что ему пришлют сообщение ***mn+1***. Поэтому Luthi дополнительно ограничивает время блокировки процесса средним промежутком времени между двумя успешно полученными сообщениями.

Для нахождения компромисса между тупиком и откатом предлагается формула вычисления *среднего риска отката* (*average rollback cost*):

Обозначим время получения сообщения ***mn*** за , а среднее время между получением двух успешно обработанных сообщений (после ***n*** таких сообщений) за:

Таким образом, максимальное время блокировки составит:



Рис. . Построение функции оценки вероятности отката

Опуская некоторые вычисления, приходим к выводу, что логический процесс следует блокировать, как только:

Если выполняется условие (), то среднее время непродуктивного ожидания без блокировки окажется выше, чем потенциальное время блокировки. Условие () гарантирует, что система не попадёт в состояние тупика.

FIPA-Based Synchronization Protocol

Алгоритм предложили немецкие специалисты Dirk Pawlaszczyk и Steffen Strassburger в 2009-м году []. Особенностью данного алгоритма является тот факт, что он ориентирован специально на агентное моделирование. Авторы в своей работе описывают разработку полноценной агентной платформы, функционирующей на базе стандарта FIPA (The Foundation for Intelligent Physical Agents). FIPA представляет собой набор спецификаций стандартов и протоколов, применяемых для создания агентных моделей.

Предлагаемый алгоритм синхронизации является достаточно эффективным, поскольку он использует знания из протокола FIPA, а значит, в некоторой степени, знания о модели. Алгоритм применяет так называемое правило *Wait\_for*. Дело в том, что некоторые FIPA-протоколы взаимодействия (напр, *FIPA:request*) подразумевают коммуникацию агентов, при которой запрос требует обязательного ответа (см. рис. 12).

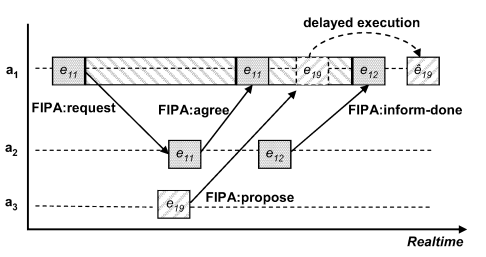


Рис. . Схема работы правила Wait\_for

Если агенту пришло такое сообщение, он может либо дать согласие на процесс коммуникации (*FIPA:agree*), либо отказаться (*FIPA:refuse*). Но в любом случае ответ должен быть отправлен обратно. При согласии агента инициатор коммуникации блокирует обработку всех новых сообщений до тех пор, пока не получит сообщение *FIPA:inform-done*.

По рис. 12 видно, что если агент ***a3*** отправляет сообщение ***e19***, то его обработка будет отложена до тех пор, пока агент ***a2*** не вышлет ответ на запрос в виде сообщения ***e12***.

Заметим, что алгоритм не сможет работать в системах, где не используется спецификация FIPA. Считать это недостатком в общем случае будет неверно, поскольку FIPA стандартизирует описание агентных моделей, в частности схему взаимодействия. А это, в свою очередь, даёт дополнительные знания, которые можно использовать для повышения эффективности синхронизации.

Lookback oriented protocols

Gilbert Chen и Boleslaw K. Szymanski предложили серию алгоритмов синхронизации, которые по определению являются оптимистическими, но по факту занимающими промежуточное положение между консервативными и оптимистическими подходами [].

Суть их идеи заключается в оценке так называемого *lookback*. Этот параметр по смыслу схож, но по времени противоположен уже известному параметру *lookahead*. Пусть ***curTime*** – локальное время логического процесса, а ***L*** – максимальное время, при котором можно обработать сообщение «из прошлого» без отправки антисообщений. Величина ***curTime - L*** называется *lookback*. , а временнóе окно [***curTime - L; curTime***] называется *lookback window*. Нижняя отметка данного окна носит название VLT (*Virtual Lookback Time*).

Ранние алгоритмы применялись для уменьшения числа каскадных откатов. Идея вполне очевидна – если сообщение «из прошлого» попадает в *lookback window*, то оно обрабатывается специальной процедурой, которая выполняет локальный откат без отправки антисообщений.

Более поздние модификации алгоритмов, основанных на lookback, вообще отказались от антисообщений, что значительно повысило качество синхронизации. Основная идея заключается в том, что каждый логический процесс обязуется выполнить так называемое *lookback-ограничение* (*lookback-constraint*). Это ограничение гарантирует, что LBTS всех будущих событий будет не меньше, чем VLT. Отсюда следует тот факт, что все будущие события обязательно попадут в *lookback-окно* любого другого процесса.

Перед обработкой события следуют выполнить предварительное вычисление VLT (*VLT pre-computing*), чтобы убедиться, что событие удовлетворяет *lookback-ограничению*. Если оно не удовлетворяет ограничению, событие не обрабатывается. Процесс ждёт до тех пор, пока LBTS не вырастет до отметки LVT.

К недостаткам данного метода можно отнести необходимость вычисления *виртуального lookback-времени*, а также необходимость обнаружения и устранения возможного тупика.

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, хотя в общем случае алгоритмы не исключают вариант, когда знания извлекаются 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:

, (2)

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 гипотезы, 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).

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*.

Алгоритм#1 служит простейшей эвристикой по устранению нежелательных откатов. Позже этот алгоритм будет объединён с Алгоритмом#3.

Определение. Сообщение относится к классу *stateless*, если его пациенс не имеет внутреннего состояния.

Утверждение. Если событие относится к классу stateless, то оно допускается к обработке «в прошлом», поскольку это не приведёт к нарушению локальной каузальности.

Доказательство. Парадоксы времени возникают из-за того, что внутреннее состояние агента (логического процесса) на момент прихода сообщения «из прошлого» не соответствует ожидаемому (т.е. состоянию на момент времени, равному временнóй метке сообщения). Если же агент не имеет внутреннего состояния, то нарушение локальной каузальности теоретически невозможно.

Таким образом, мы получаем, что агент работает как некоторый простой вычислитель функции – на один и тот же вход он выдаёт один и тот же выход.

Утверждение. Если агент обладает свойством stateless, то он является реактивным.

Утверждение в обратную сторону неверно – если агент реактивный, то не факт, что он будет обладать свойством stateless, а значит, будет безопасным в смысле локальной каузальности.

Итак, алгоритм представляет собой всего лишь 1 условный оператор:

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

**if** (event isStateless)

handleEvent(event)

**else** rollback(event)

Приведённый выше код является упрощённым. Позже в разделе «Алгоритм#3» будет приведён подробный фрагмент обработчика сообщений, включающий Алгоритм#1.

Нужно отметить, что предложенный алгоритм будет прекрасно работать во многих простых моделях с реактивными агентами. Но по мере усложнения модели агенты будут развиваться – у них может появиться память, знания и механизмы вывода на знаниях, а значит, практическая ценность Алгоритма#1 будет снижаться.

В качестве примера stateless-агента можно предложить некоторый простой сервис – например, моделирование кассира на автозаправке, который отвечает на запросы, но не хранит внутренних переменных.

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.

Какие можно сделать выводы из данного утверждения? Существует, как минимум, два варианта 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 можно сформулировать 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. 10).



Рис. . Проблема взаимной блокировки

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 подробно проиллюстрирована на рис. 11.



Рис. . Возникновение бесконечного ожидания

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 *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 событий 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 события 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?

Цель. 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*».

(3)

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

(4)

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.

(5)

Подставляя выражение для Twhile в Tsafe, получаем:

(6)

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.

(7)

Запишем теперь общее рекуррентное соотношение:

(8)

Данную формулу в целом 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:

(9)

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

(10)

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

(11)

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

Тогда рекуррентное выражение принимает следующий вид:

(12)

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

(13)

Следует отметить справедливость следующего выражения:

(14)

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.

Трудности в реализации алгоритма

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 6. Отметим, что в реализации был выбран именно этот подход как наиболее прогрессивный.
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 22.05.2013г.

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 программного обеспечения.

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 этом 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.

Трейты

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 языков). Примесь – это элемент языка программирования, реализующий некоторое поведение, которое можно затем «подмешать» (*to mixin*) к некоторому существующему элементу.

В языке Scala в роли такой функциональной единицы выступает *трейт* (Trait). Примеси впервые были использованы в языках *Flavors* и *CLOS*. Среди современных языков в чистом виде поддерживаются в *Ruby*, *Scala, PHP* и *Python*.

Основное отличие трейтов от интерфейсов – завершённость. Хотя трейты и могут иметь абстрактные члены, всё же на практике большинство полей, типов и методов полностью определены. Это объясняется тем, что концептуально трейты добавляют поведение, а не предписывают его, как это делают интерфейсы.

Ниже приведён простой пример «примешивания» поведения:

**class** Animal {

**def** *move*() {...} // животные умеют передвигаться

}

**trait** Rodent {

**def** *nibble*() {...} // грызуны умеют грызть

}

**class** Mouse **extends** Animal **with** Rodent {

**def** *test*() { // итоговый класс наследует оба поведения

*move*()

*nibble*()

}

}

Структуры данных, используемые в системе

К основным типам данных, применяемым в системе, относят события и сообщения. Также существует понятие *категории* (см. ниже раздел «*Трейт ModelObservable*»).

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 с предыдущими реализациями платформы.

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. 12.



Рис. . Многослойная архитектура симулятора

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 компоненты, расширяющие структуру, поведение и семантику модуля.

В последующих параграфах мы рассмотрим, как «подмешивание» поведения помогает структурно наращивать функциональные возможности симулятора.

Трейт Simulator

Трейт *Simulator* (рис. 13) является базовым модулем, к которому подмешиваются другие элементы, уточняя его поведение. Фактически, *Simulator*, – это каркас для логического процесса.



Рис. . Структура трейта Simulator

В симуляторе два 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 функцию *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.

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.

Трейт Communicator

Трейт *Communicator* (рис. 14) 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]



Рис. . Структура трейта Communicator

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

*OptimisticSynchronizator* (рис. 15) 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]



Рис. . Структура трейта OptimisticSynchronizator

**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.

Синхронизатор добавляет одну полезную пользовательскую функцию – *stopModelling*.

Трейт ModelObservable

Данный трейт (рис. 16) 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 реальной реализации 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.



Рис. . Структура трейта ModelObservable

**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* (рис. 17) 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]



Рис. . Структура трейта Analyser

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, Kotlin 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*[[4]](#footnote-4), 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[[5]](#footnote-5). 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*;
* определить 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

В данной главе будет рассмотрен онтологический подход к представлению знаний, используемых оптимистическими алгоритмами для повышения эффективности синхронизации агентов. В главе будет показано, как онтология может быть декомпозирована на обобщённую (предметную) онтологию и онтологию задачи. Также приводится пример построения онтологии задачи.

Онтологический подход к представлению знаний о модели

Применение знаний в программных системах является, пожалуй, ключевым отличием интеллектуальных систем от обычных прикладных программ (такие системы зачастую называют СБЗ – системы, базирующиеся на знаниях). И несмотря на то, что точного определения *систем искусственного интеллекта* не существует[[6]](#footnote-6), СБЗ применяются во многих сферах для решения многих задач, в т.ч. для тех, для которых формализация затруднена или невозможна.

Мультиагентные системы являются одним из наиболее ярких представителей систем ИИ (в традиционной классификации искусственного интеллекта они занимают промежуточное положение между когнитивной и бионической ветвями). Интеллектуальность придаёт агентам свойство решать задачу «нестандартным» образом, когда программные сущности вместо традиционных вычислений используют коммуникацию и вывод на знаниях.

Мы имеем дело с агентным моделированием, которое, в некоторой степени, является своего рода мультиагентной системой – множество агентов совместно решают задачу имитационного моделирования, оценивая ситуацию, общаясь между собой и принимая некоторые решения. Поэтому в силу своей специфики агентное моделирование немедленно предполагает наличие знаний у агентов, на базе которых они могут принимать решения, выводить новые факты и утверждения и обмениваться этими фактами друг с другом.

Что касается алгоритмов синхронизации, то в большинстве рассматриваемых работ (см. главу 4) авторы утверждают, что использование знаний о модели является «плохим тоном» в PDES. Объясняется это отсутствием универсальности в используемых алгоритмах: хороший алгоритм должен выполнять надёжную синхронизацию независимо от состояния задачи и предметной области. В связи с этим, авторы всегда рассматривают абстрактные логические процессы в абстрактной предметной области, которые обмениваются сообщениями произвольной природы.

Пермская школа распределённого имитационного моделирования совершила революцию в области оптимистической синхронизации дискретно-событийных моделей! Уже в 2009-м году появляется первая работа (авторы: Ермаков, Замятина []), ставящая под сомнение необходимость отказа от знаний при синхронизации логических процессов. Авторы применяют специальную СБЗ, представленную в виде продукционной экспертной системы, которая управляет сдерживанием оптимизма.

Действительно, даже классические алгоритмы явным или неявным образом полагаются на «знания» о модели: *lookahead* в консервативных алгоритмах, *lookback* в оптимистических, циклы в алгоритме *Work Flow* и т.д. Данная работа предлагает подход, целиком противоположный мнению большинства зарубежных авторов: мы ***будем*** использовать знания о модели с целью повышения эффективности синхронизации времени.

Гипотеза. Применение знаний о модели повышает эффективность оптимистических алгоритмов синхронизации логических процессов (напомним, что под эффективностью мы понимаем количество откатов и общее время моделирования).

Знания о предметной области представлены в виде онтологии. Данное понятие будем рассматривать в общепринятом определении:

Определение. *Онтология предметной области* – это тройка {*T, R, Ф*},

где *T* – множество понятий из предметной области;

*R* – множество связей между понятиями;

*Ф* – множество функций интерпретации понятий из *T* посредством связей из *R.*

Определение не специфицирует сущность понятий, типы связей и представление функций интерпретации, но в общем случае предполагается лишь одно ограничение:

В зависимости от мощности множеств R и Ф можно выделить различные формы онтологий: словари, глоссарии, таксономии понятий, тезаурусы и т.д. Существуют и другие классификации онтологий по различным признакам, но общая их задача остаётся одна – специфицировать знания о некоторой предметной области в явном виде.

Предметная онтология и онтология задачи

Согласно одной из бесчисленного множества классификаций, онтологии можно разделить на 3 категории:

* метаонтология;
* предметная онтология;
* онтология задач.

Метаонтология оперирует общими понятиями, которые в целом не зависят от предметной области. Предметная онтология, очевидно, описывает понятия из предметной области. Онтология задач возникает в так называемой *проблемной области* (когда имеется предметная область и поставленная задача).

В рамках исследовательской работы было принято решение разработать предметную онтологию, которая могла бы описать предметную область «Оптимистическая синхронизация в агентном PDES-моделировании». Исследователю, тем самым, открывается возможность быстро создавать онтологии задач, соединяя понятия из конкретной модели (напр, *Перевозчик*) с понятиями из предметной области (напр, *Агент*).

Ниже представлены основные этапы разработки онтологии. Следует обратить внимание, что многие понятия совпадают с теми, что были представлены в предыдущей главе. Это свидетельствует о том, что концепты, описываемые в онтологии, могут быть легко представлены в виде объектов при объектно-ориентированном проектировании.

При декомпозиции предметной области следует обратить внимание на 3 важных составляющих:

* *агентное моделирование* – машинная имитация выполняется агентами;
* *дискретно-событийное моделирование* – вводится понятие события;
* *оптимистическая синхронизация* – во-первых, система параллельная, во-вторых, применяются оптимистические алгоритмы синхронизации.

Общий вид предметной онтологии проиллюстрирован на рис. 18 и рис. 19. Оперируя теорией множеств, можно сказать, что предметная область представлена тремя множествами:

* множество агентов;
* множество событий;
* множество функций преобразования модели.

Отметим, что имеется вариант сделать функции преобразования частью агентов. Такой вариант имеет право на существование, но в нашем случае эти множества разделены.



Рис. . Предметная онтология. Слой пользователя



Рис. . Предметная онтология. Слой разработчика симулятора и алгоритмов синхронизации

Как мы видим, онтология имеет многослойную структуру. Связано это с тем, что основной практической ценностью онтологии является возможность использования её в качестве основы для создания онтологии задачи. Поскольку онтологию задачи составляет исследователь модели, он должен получить доступ ко всем необходимым концептам из предметной онтологии, и только к ним. Такие понятия, как *Сообщение*, *Антисообщение* и др. должны быть скрыты от него.

Фактически исследователь будет использовать понятие *События*, которое несёт в себе временную метку (*timestamp*) и *Тип События*. Тип события, в свою очередь, определён тройкой *Агенс-Пациенс-Предикат*, которые, как видно из рис. 18, ссылаются на *Агентов* и *Функции Преобразования*.

Следующий слой описывает концепты для разработчика симулятора: ему уже требуются такие знания, как *Локальный* и *Удалённый Агент*, а также *Сообщение*. Нетрудно заметить, что данный слой расширяет предыдущий – здесь агенты классифицируются по принадлежности локальному процессу. Более того, онтология специфицирует тот факт, что удалённые агенты упаковывают события внутрь сообщений.

Наконец, слой алгоритмов синхронизации оперирует наиболее сложными понятиями, которые должны быть скрыты как от исследователя, так и от разработчика симулятора: *Откаты*, *Антисообщения* и т.п. Рис. 19 показывает, что сообщения делятся на «позитивные» и антисообщения, причём первые из них могут быть *Regular* (не вызывают откаты) и *Struggling* (вызывают откаты). Онтология даёт им определения в явном виде:

Struggling = m: Message => m.event.timestamp < current\_time

Regular = m: Message => m.event.timestamp ≥ current\_time

Также имеется одно ограничение:

∀m: Antimessage ∃ p: Positive message: m.id = p.id

В программной платформе «Better» онтология представлена в виде отдельного подключаемого компонента. Для хранения используется текстовый формат OWL стандарта OWL-DL.

Пример онтологии тестовой модели

В данном разделе мы приведём в качестве примера фрагмент онтологии задачи для модели «Супермаркет», которая будет подробно описана в следующей главе. Здесь мы сделаем акцент именно на методике построения онтологии.

Пример онтологии в графическом виде изображён ниже на рис. 19.



Рис. . Фрагмент онтологии тестовой модели «Супермаркет»

В центре рисунка изображено событие ***requestToSmoke***. В соответствии с *условием семантической определённости* для него обязательно должны быть определены агенс, пациенс и предикат. В роли первого выступает агент *Охранник* (*Guard*), в роли второго – *Кассир2* (*Cashier2*), в качестве предиката – ссылка на некоторую функцию (обозначена как *f5*).

Дополнительные связи вводятся для повышения семантической мощности механизмов синхронизации агентов. В нашем случае это отношение ***waits\_for***, которое непосредственно использует Алгоритм#2. При наличии у события такого атрибута логический процесс при необходимости заблокирует свою работу.

Отношение ***produces*** представляет собой ссылку на список тех событий, которые могут быть сгенерированы при обработке текущего события. Это исключает необходимость «псевдообработки», что автоматически исключает дополнительные вычисления и устраняет проблемы, связанные с генераторами случайных чисел (см. главу 4, раздел «Сложности в реализации алгоритмов»). В следующем разделе покажем, как применение отношения ***produces*** изменяет теоретическую оценку сложности алгоритмов.

Теоретическая оценка сложности с учётом знаний о модели

В главе 4 были получены теоретические оценки сложности по времени (формулы (8), (9), (10), (13)). 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.

С применением знаний о модели, сложность генерации потомков событий сводится к функции обращения к подсистеме знаний, а значит, имеет место равенство:

(15)

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

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

(16)

***Минимальная оценка:***

Минимальная оценка совпадает с (9):

***Максимальная оценка:***

(17)

***Средняя оценка:***

Также полагаем, что справедливо неравенство (14):

Предложенные теоретические оценки уже можно считать итоговыми в рамках исследовательской работы.

1. Эксперименты. Практические результаты

Данная глава является заключительной. В ней представлена тестовая модель для экспериментальной проверки корректности алгоритмов, описанных в главе 4 на базе разработанной платформы, представленной в главе 5 на основе знаний о модели, которые представлены способом, описанным в главе 6. Тестовая модель «Супермаркет» служит только для анализа алгоритмов и не несёт предметно-исследовательской ценности.

Общее описание модели

Тестовая агентная модель «Супермаркет» разрабатывалась специально для практического анализа алгоритмов синхронизации. Тип модели – СМО (*Системы массового обслуживания*).

Определение. *Система массового обслуживания* – математическая модель, описывающая систему, в которой обслуживающие устройства обрабатывают некоторый (дискретный или непрерывный) поток заявок.

Отличительными особенностями модели являются:

* относительная простота реализации;
* интенсивная коммуникация между агентами.

Эксперименты проводились на 2-х логических процессах, однако сама модель легко масштабируется на множество процессов. Интенсивная коммуникация благоприятствует частому нарушению локальной каузальности и, как следствие, многочисленным откатам.

Итак, модель представляет собой совокупность 5 агентов:

Guard (Охранник супермаркета)

* может открыть/закрыть двери супермаркета;
* может выходить покурить;
* может позвать Кассира2 покурить;
* может отлучиться на технологический перерыв;
* может заподозрить посетителя в краже товара.

Cashier1 (Кассир1)

* обслуживает покупателей;
* может отлучиться на технологический перерыв.

Cashier2 (Кассир2)

* обслуживает покупателей;
* может отлучиться на технологический перерыв;
* может выйти с Охранником покурить.

Purchaser (Покупатель)

* выбирает товары;
* может своровать товар;
* может выбрать кассу для обслуживания;
* может выбрать способ оплаты (наличный/безналичный расчёт).

Supermarket (Окружающая среда)

Окружающая среда представляет собой реактивного агента, хранящего общее состояние модели.

Агент покупателя присутствует в модели во множестве экземплярах, остальные агенты – в единственном экземпляре.

Цель модели: получить информацию об эффективности работы персонала (средняя длина очереди и объёмах похищенного товара).

Напомним, что наша цель отличается от цели модели – нам требуется сравнить эффективность алгоритма Time Warp и алгоритмов, основанных на знаниях об агентной модели. Под *эффективностью*, напомним, будем понимать:

* общее количество откатов;
* совокупное время моделирования.

Формализация агентной модели

Далее в разделе будем предполагать, что генератор случайных чисел *random* формирует значение случайной величины по равномерному закону распределения.

Guard

* **Инициализация**
  + Событие:
    - Время: 5[[7]](#footnote-7);
    - Пациенс: Supermarket;
    - Предикат: Открыть дверь.
  + Событие:
    - Время: 25 + random(15);
    - Пациенс: Cashier2;
    - Предикат: Предложить покурить.
  + Событие:
    - Время: 65 + random(9);
    - Пациенс: Guard;
    - Предикат: Технологический перерыв.
* **Ответ Кассира2**
  + Событие:
    - Время: t + 15 + random(15);
    - Пациенс: Cashier2;
    - Предикат: Предложить покурить.
  + Условие: C1
    - ответ = «да» | | random(1)>0.5.
  + Событие:
    - Условие: C1;
    - Время: t + random(1);
    - Пациенс: Supermarket;
    - Предикат: Охранник покинул рабочее место.
  + Событие:
    - Условие: C1;
    - Время: t + 2 + random(7);
    - Пациенс: Guard;
    - Предикат: Охранник вернулся.
* **Технологический перерыв**
  + Событие:
    - Время: t + random(1);
    - Пациенс: Supermarket;
    - Предикат: Охранник покинул рабочее место.
  + Событие:
    - Время: t + 1 + random(3);
    - Пациенс: Guard;
    - Предикат: Охранник вернулся.
* **Охранник вернулся**
  + Событие:
    - Время: t + random(1);
    - Пациенс: Supermarket;
    - Предикат: Охранник занял рабочее место.

Cashier1

* **Переменные**
  + *Вдохновение* = {***true***, ***false***}.
* **Инициализация**
  + Событие:
    - Время: 75 + random(10);
    - Пациенс: Cashier1;
    - Предикат: Технологический перерыв.
* **Технологический перерыв**
  + Событие:
    - Время: t + random(1);
    - Пациенс: Supermarket;
    - Предикат: Кассир1 покинул рабочее место.
  + Событие
    - Время: t + 5 + random(4);
    - Пациенс: Cashier1;
    - Предикат: Кассир1 вернулся.
* **Кассир1 вернулся**
  + Событие:
    - Время: t + random(1);
    - Пациенс: Supermarket;
    - Предикат: Кассир1 занял рабочее место.
* **Обслужить покупателя (по наличному/безналичному расчёту)**
  + *Вдохновение* = !*Вдохновение.*
  + Событие:
    - Время: t + random(1);
    - Пациенс: Supermarket;
    - Предикат: Совершить оплату.
  + Событие:
    - Время: t + 1.5 + random(*Вдохновение* ? 1 : 3);
    - Пациенс: агенс;
    - Предикат: Платёж принят.

Cashier2

Агент является подклассом агента *Cashier1*. Он копирует существующие предикаты (заменяя при необходимости постфиксы с «1» на «2») и добавляет один новый предикат:

* **Предложение покурить**
  + Событие:
    - Время: t + 1;
    - Пациенс: Guard;
    - Предикат: Ответ Кассира2.
  + Условие: C2
    - random(1) > 0.4;
  + Событие:
    - Условие: C2;
    - Время: t + random(1);
    - Пациенс: SuperMarket;
    - Предикат: Кассир2 покинул рабочее место.
  + Событие:
    - Условие: C2;
    - Время: t + 2 + random(7);
    - Пациенс: Cashier2;
    - Предикат: Кассир1 вернулся.

Purchaser

* **Переменные**
  + *Склонность\_к\_воровству* = random(1)/2.
* **Инициализация**
  + Событие:
    - Время: 5 + random(30);
    - Пациенс: агенс;
    - Предикат: Вход в супермаркет.
* **Вход в супермаркет**
  + *Склонность\_к\_воровству* = random(1)/2;
  + Событие:
    - Время: t + random(1);
    - Пациенс: Supermarket;
    - Предикат: Обновить данные о числе покупателей.
  + Событие:
    - Время: t + random(1);
    - Пациенс: Supermarket;
    - Предикат: Обновить данные об общем числе покупателей.
  + Событие:
    - Время: t + 1 +random(5);
    - Пациенс: агенс;
    - Предикат: Выбор товара.
* **Выбор товара**
  + Условие: C3
    - random(1) > 0.8.
  + Условие: C4
    - random(1) ≤ 0.8.
  + Условие: C5
    - random(1)\**Склонность\_к\_воровству* > 0.35.
  + Событие:
    - Условие: C3;
    - Время: t + 1 + random(5);
    - Пациенс: агенс;
    - Предикат: Покупатель пошёл на кассу.
  + Событие:
    - Условие: C4;
    - Время: t + 1 + random(5);
    - Пациенс: агенс;
    - Предикат: Выбор товара.
  + Событие:
    - Условие: C5;
    - Время: t + random(1);
    - Пациенс: Supermarket;
    - Предикат: Увеличить счётчик краж.
  + Событие:
    - Условие: C5;
    - Время: t + 1;
    - Пациенс: Guard;
    - Предикат: Заподозрить покупателя.
* **Покупатель пошёл на кассу**
  + Событие:
    - Время: t + random(1);
    - Пациенс: random(1)>0.5 ? Cachier1 : Cashier2;
    - Предикат: random(1)>0.5 ? Обслужить по наличному расчёту: Обслужить по безналичному расчёту.
* **Платёж принят**
  + Событие:
    - Время: t + random(1);
    - Пациенс: SuperMarket;
    - Предикат: Уменьшить число покупателей.
  + Событие:
    - Время: t + 5 + random(30);
    - Пациенс: агенс;
    - Предикат: Вход в супермаркет.

Результаты экспериментов

Моделирование проводилось в кластере на 2-х вычислительных узлах. На первом логическом процессе располагались *Охранник*, *Кассир1* и 8 агентов-покупателей. На втором были размещены соответственно *Кассир2*, *Супермаркет* и 8 других агентов-покупателей.

Модель была выполнена многократно при одних и тех же параметрах. Общая длительность моделирования составляла 300 (±15) ед. модельного времени.

Результаты тестирования алгоритма Time Warp представлены ниже в таблице 2. Среднее время имитационного прогона составляет 134.24 сек.

Таблица . Результаты работы алгоритма Time Warp на тестовой модели

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Категория** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Среднее** |
| **EVENTS\_HANDLED:** | 1197 | 1197 | 1229 | 1229 | 1229 | 1259 | 1289 | 1229 | 1259 | 1199 | **1231,6** |
| **MAX\_TIME\_WINDOW:** | 10 | 8 | 13 | 17 | 9 | 14 | 19 | 10 | 10 | 9 | **11,9** |
| **RECEIVED\_ANTI\_MESSAGES:** | 9 | 12 | 8 | 5 | 13 | 6 | 9 | 6 | 8 | 8 | **8,4** |
| **RECEIVED\_EVENT\_MESSAGES:** | 76 | 89 | 91 | 82 | 89 | 80 | 88 | 85 | 92 | 87 | **85,9** |
| **RECEIVED\_MESSAGES:** | 85 | 101 | 99 | 87 | 102 | 86 | 97 | 91 | 100 | 95 | **94,3** |
| **ROLLBACKS:** | 61 | 67 | 77 | 73 | 74 | 70 | 74 | 69 | 74 | 73 | **71,2** |
| **ROLLBACKS\_DEPTH\_1:** | 12 | 10 | 15 | 11 | 9 | 9 | 7 | 11 | 12 | 13 | **10,9** |
| **ROLLBACKS\_DEPTH\_2:** | 5 | 12 | 10 | 14 | 18 | 9 | 9 | 10 | 10 | 10 | **10,7** |
| **ROLLBACKS\_DEPTH\_3:** | 11 | 8 | 17 | 9 | 10 | 12 | 8 | 11 | 8 | 15 | **10,9** |
| **ROLLBACKS\_DEPTH\_4:** | 12 | 10 | 13 | 6 | 8 | 10 | 12 | 5 | 15 | 11 | **10,2** |
| **ROLLBACKS\_DEPTH\_5:** | 3 | 9 | 8 | 10 | 5 | 11 | 9 | 8 | 7 | 10 | **8** |
| **ROLLBACKS\_DEPTH\_6:** | 6 | 7 | 8 | 5 | 5 | 6 | 7 | 13 | 9 | 6 | **7,2** |
| **ROLLBACKS\_DEPTH\_7:** | 4 | 5 | 3 | 6 | 7 | 6 | 8 | 2 | 4 | 3 | **4,8** |
| **ROLLBACKS\_DEPTH\_MORE:** | 20 | 16 | 18 | 23 | 21 | 16 | 21 | 20 | 21 | 18 | **19,4** |
| **ROLLBACKS\_MAXDEPTH:** | 21 | 18 | 24 | 24 | 27 | 18 | 32 | 16 | 22 | 16 | **21,8** |
| **SENT\_ANTI\_MESSAGES:** | 103 | 79 | 115 | 110 | 116 | 105 | 139 | 101 | 122 | 94 | **108,4** |
| **SENT\_EVENT\_MESSAGES:** | 416 | 386 | 438 | 412 | 433 | 459 | 466 | 424 | 442 | 409 | **428,5** |
| **SENT\_MESSAGES:** | 519 | 465 | 553 | 522 | 549 | 564 | 605 | 525 | 564 | 503 | **536,9** |

А ниже представлены результаты работы алгоритмов, основанных на знаниях (таблица 3). Среднее время имитационного прогона составляет 116.12 сек.

Таблица . Результаты работы алгоритмов KBASA на тестовой модели

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Категория** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Среднее** |
| **EVENTS\_HANDLED:** | 735 | 941 | 916 | 859 | 883 | 871 | 820 | 870 | 915 | 947 | **875,7** |
| **MAX\_TIME\_WINDOW:** | 2 | 3 | 5 | 8 | 15 | 6 | 14 | 6 | 2 | 7 | **6,8** |
| **RECEIVED\_ANTI\_MESSAGES:** | 3 | 2 | 0 | 1 | 0 | 1 | 0 | 3 | 1 | 0 | **1,1** |
| **RECEIVED\_EVENT\_MESSAGES:** | 71 | 89 | 73 | 75 | 71 | 80 | 71 | 88 | 83 | 86 | **78,7** |
| **RECEIVED\_MESSAGES:** | 74 | 91 | 73 | 76 | 71 | 81 | 71 | 91 | 84 | 86 | **79,8** |
| **ROLLBACKS:** | 1 | 3 | 5 | 4 | 5 | 2 | 7 | 4 | 1 | 6 | **3,8** |
| **ROLLBACKS\_DEPTH\_1:** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| **ROLLBACKS\_DEPTH\_2:** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| **ROLLBACKS\_DEPTH\_3:** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | **0,1** |
| **ROLLBACKS\_DEPTH\_4:** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| **ROLLBACKS\_DEPTH\_5:** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | **0,1** |
| **ROLLBACKS\_DEPTH\_6:** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | **0,2** |
| **ROLLBACKS\_DEPTH\_7:** | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0,2** |
| **ROLLBACKS\_DEPTH\_MORE:** | 0 | 1 | 5 | 4 | 4 | 2 | 7 | 3 | 1 | 5 | **3,2** |
| **ROLLBACKS\_MAXDEPTH:** | 7 | 9 | 14 | 25 | 27 | 18 | 36 | 22 | 11 | 20 | **18,9** |
| **SENT\_ANTI\_MESSAGES:** | 1 | 4 | 17 | 16 | 20 | 8 | 30 | 11 | 2 | 19 | **12,8** |
| **SENT\_EVENT\_MESSAGES:** | 262 | 353 | 344 | 316 | 337 | 331 | 273 | 324 | 330 | 349 | **321,9** |
| **SENT\_MESSAGES:** | 263 | 357 | 361 | 332 | 357 | 339 | 303 | 335 | 332 | 368 | **334,7** |

Как мы видим, время моделирования сократилось на 18 секунд. Результат не совсем большой, однако, это объясняется простотой модели (а именно, небольшой сложностью вычислений). На более сложных моделях разница будет более заметна.

А теперь рассмотрим другие показатели (напомним, что из расчёта на 300 ед. модельного времени):

* Количество откатов сократилось с 71.2 до 3.8-х (рис. 20)! Потрясающий результат! Это, наряду с общим временем моделирования, подтверждает эффективность новых алгоритмов синхронизации.
* Общее число неглубоких откатов сократилось практически до нуля (в среднем 0.1-0.2 против 7-11). Это, в основном, демонстрирует работу Алгоритма#3 – чем меньше глубина цепочки отката, тем больше вероятность того, что все события из неё будут каузально независимы. А значит, на коротких цепочках откаты будут происходить весьма редко.
* Число глубоких откатов также сократилось с 19.4 до 3.2 (рис. 22).
* Количество отправленных антисообщений сократилось с 108.4 до 12.8 (рис. 21). Этот эффект, очевидно, коррелирует с уменьшением числа откатов.
* Общее количество отправленных сообщений также уменьшилось (с 536.9 до 334.7). Результат показательный, даже несмотря на то, что новые алгоритмы требуют отправки дополнительных служебных сообщений (*TimeRequest*, *TimeResponse*), которые также суммируются.
* Общее количество обработанных событий уменьшилось с 1231.6 до 875.7. Это означает, что в среднем алгоритм Time Warp как минимум 350 событий обрабатывал зря (они были утрачены ввиду откатов).
* Общее количество принятых сообщений также сократилось вместе с сокращением отправленных сообщений (с 94.3 до 79.8).
* Максимальная разница «возвращённого назад» времени сократилась практически вдвое: с 11.9 до 6.8.

Рис. . Количество откатов за 300 ед. модельного времени

Рис. . Отправлено антисообщений за 300 ед. модельного времени

Рис. . Число глубоких откатов за 300 ед. модельного времени

Итак, мы видим, что алгоритмы синхронизации агентов, основанные на знаниях, действительно подтверждают свою эффективность на практике. Единственная неприятность: максимальная глубина отката сократилась незначительно (с 21.2 до 18.1). Для решения проблемы можно воспользоваться уже существующими эвристиками, например, стратегией *Moving Time Windows*, когда величина «оптимизма» ограничивается сверху некоторой константой – размером временнóго окна.

Заключение

Библиографический список

Глоссарий

1. *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.
2. *Локальная каузальность* – гарантия того, что все события в ходе имитационного прогона будут выполнены по возрастанию (неубыванию) временных меток.
3. *LBTS* (от англ. *Lower Bound of TimeStamps* – нижняя граница временных меток) – время, равное минимуму временных меток по всем входным очередям сообщений.
4. *Нулевое сообщение* – сообщение, которое (семантически) несёт в себе данные только о временнóю метке.
5. *Антисообщение* – сообщение, отменяющее ранее высланное сообщение.
6. *Каскадный откат* – откат, порождённый принятым от другого логического процесса антисообщением.
7. *Порядок каскадного отката* – число элементов в цепочке каскадного отката.
8. *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 каскадных откатов.
9. *Терминальное событие* – событие, обработка которого не порождает новых событий.
10. *Цепочкой отката* называют последовательность событий (и/или состояний модели), упорядоченную по убыванию временных меток.
11. *Блокирующее событие* – событие, для которого выполняются 2 условия:

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

1. *Разблокирующее событие* – событие, содержащие ответ на блокирующее событие.
2. *Контрарная пара* – совокупность блокирующего и разблокирующего события.
3. 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.
4. *Stateless-событие* – событие, пациенс которого не имеет внутреннего состояния.
5. *Информационные процедуры* – программные компоненты, накладываемые на модель с целью сбора статистики в ходе имитационного прогона.
6. *Онтология предметной области* – это тройка {*T, R, Ф*},

где *T* – множество понятий из предметной области;

*R* – множество связей между понятиями;

*Ф* – множество функций интерпретации понятий из *T* посредством связей из *R.*

1. *Система массового обслуживания* – математическая модель, описывающая систему, в которой обслуживающие устройства обрабатывают некоторый (дискретный или непрерывный) поток заявок.

Приложение

1. Понятно, что на практике оно может содержать вспомогательные данные [↑](#footnote-ref-1)
2. По мнению автора данной работы, наличие дополнительных знаний о модели является не ограничением, а наоборот, преимуществом (см. следующую главу). [↑](#footnote-ref-2)
3. Не путать с GVT – Global Virtual Time [↑](#footnote-ref-3)
4. 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-4)
5. В Java существуют более эффективные методы блокировки: *wait*, ReentrantLock, Semaphore и др. В нашем примере для простоты используется простая булевская переменная. [↑](#footnote-ref-5)
6. Обычно под системами ИИ понимают такие системы, которые решают задачи, свойственные (ввиду отсутствия формального представления) для решения только человеком [↑](#footnote-ref-6)
7. ед. модельного времени [↑](#footnote-ref-7)