Поведение персонажа в играх

Автор: Джефф Оркин, Тайнан Смит, Деб Рой
Перевод: Карпунов Геннадий
Источник: http://www.aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/viewFile/2115/2562

Аннотация

Чтобы эффективно взаимодействовать с игроком, персонаж должен найти цели и принять соответствующие ответные меры. Этот процесс разбора входного потока для создания вывода об иерархической структуре задачи такой же, как и процесс компиляции исходного кода. Мы проведем аналогию между компиляцией исходного кода и компиляцией поведения, и предложим моделирование когнитивной системы персонажа как компилятор, который разбивает замечания и выводит иерархическую структуру задачи. Оценки сравнения атоматически преобразуют поведение в человеческие повадки, что демонстрирует потенциал для этого подхода, который позволил персонажам ИИ понять поведение и вывести задачи человеческа.

Вступление

Добыча данных от игроков предлагает новые перспективные решение для авторизации персонажа в играх. Этот экспериментальный подход доказал свою эффективность при создании игроков, управляемых компьютером, для стратегий реального времени (Ortanon и др. 2007; Weber & Mateas 2009), и предварительные результаты свидетельствуют о наличии потенциала для создания поведения по данным играм (Orkin & Roy 2009). Хотя это просто записывать игры в лог-файл с отметкой времени действия, изменения состояния, и текст чата, остается открытым вопрос, как лучше всего обрабатывать и представлять эти данные так, чтобы они были полезны для персонажа ИИ. Многие игры сегодня реализуют поведение персонажа с использованием иерархических представлений, таких, как варианты иерархического планирования или иерархических конечных автоматов (Gorniak & Davis 2007; и др.. Hoang 2005 года; Fu и Houlette 2004), например, иерархические сети задач в Killzone 2 (Straatman 2009), или деревья поведения в серии Halo (Isla 2005). В идеале, мы хотели бы превратить последовательность наблюдений, записанных в логи, в один из этих знакомых иерархических структур. Этот процесс разбора входного потока символов для создания иерархической структуры очень похож на процесс компиляции в исходный код (Aho и др.. 1986). Мы компилируем код по нескольким причинам - для проверки синтаксиса, для компактного размер базы кода, а также для представления программы, такой, чтобы она могла быть легко выполнена на машине. Теми же причинами мотивируется компиляции поведения для персонажей в играх. Наблюдая взаимодействия других человеческих и/или ИИ агентов, персонажу нужно разобрать эти наблюдения признать действительным поведение, и отдельный сигнал от шума. Так как мы собираем тысячи (или даже миллионы) логов, хранение их в памяти не может быть проигнорировано. Хотя централизованная система может выполнять действия иерархического плана по контролю группы ИИ персонажей, если команда включает в себя игроков-людей, централизованный контроль уже не подходит. Для того, чтобы сотрудничать эффективно с людьми (на поле боя, или в социальных эпизодах, как ресторан) персонажу необходимо понять поведение других, выведение задачи, которые они преследуют, и выполнить соответствующие ответные меры в правильном контексте. В данной работе мы проводим аналогию между компиляцией исходного кода и компиляцией поведения. Мы предлагаем моделировать когнитивную систему персонажа, как компилятор, который разбивает замечания и выводит структуру иерархической задачи. Такая структура дает контекст для понимания поведения других, и

для выбора контекстуально соответствующих мер в ответ. Однако, аналогия не является совершенной. Существуют значительные различия между компиляцией кода и компиляцией поведения. Поведение существует в шумном помещении, где несколько персонажей могут выполнять несколько, возможно, противоречивых целей, или участвовать в поисковом поведении, которое не способствует общей цели. Компилятор исходного кода обрабатывает код для одной программы, и завершается с ошибкой, когда он находит неверный синтаксис. Компиляция поведения, с другой стороны, требует способность игнорировать фрагменты неверного синтаксиса не останавливаясь, и одновременно обрабатывать запутанные потоки замечаний, которые могут способствовать различным целям. Покажем, эти идеи, описывая работу с данными из игры Ресторан, содержащую почти 10000 записей логов взаимодействия между официантками и клиентами. Для начала мы расскажем, как мы разбиваем действия, изменения состояния, и текст чата в лексикон. Далее мы подробно расскажем, как эти знаки могут быть скомпилированы в модель поведения во время выполнения, или в качестве предварительной обработки шага. Наконец, мы оценим, насколько хорошо наша система компилирует поведение по сравнению с человеческой, и относится к нашей предыдущей работе.

Игра Ресторан

Ресторан - онлайн игра, в которой люди анонимно в парах играют роли клиентов и официанток в виртуальном ресторане. Игроки могут общаться печатным текстом, перемещаться в 3D сцене, и манипулировать 47 движениями: забрать, положить вниз, дать, проверьте, сидеть, есть, и дотронуться. Объекты реагируют на эти действия по-разному - пища уменьшается от укуса к укусу, когда едят, во время еды кресло издает звук, но не меняет форму. Шеф-повар и бармен жестко закодированы на производство продуктов питания на основе ключевых слов в текст чата. Игра занимает около 10-15 минут. Типичная игра состоит из 84 физических действий и 40 высказываний. Все игроки то и делают, что создают логи на наших серверах. Взаимодействие игроков сильно различается, от драматизации, до заказа вишневого пирога. Подробная информация о первой итерации нашей системы планирования для ИИ-контролируемых персонажей доступны в предыдущей публикации (Orkin & Roy 2009).

Figure 1. Screenshot from *The Restaurant Game*.

Лексический анализ

Чтобы собрать последовательность наблюдения, для начала необходимо разбить входной поток. При компиляции исходного кода, лексический анализатор, такой как инструмент Unix Lex занят регулярными выражениями на входе и экспортированием маркеров на их место (Levin и др.. 1992). Крупицу данных может представлять один символ на входе, или шаблон, состоящий из нескольких соседних символов. Например, Lex спецификацию для анализа кода Java может включать:

```
"if"      { return IF; }
"else"    { return ELSE; }
"while"   { return WHILE; }
"="       { return EQ; }
"!="      { return NE; }
[0-9]+    { return NUMBER; }
```

Руководствуясь этой спецификацией, Лекс создает программу, которая преобразует человеческий исходный код в последовательность маркеров, которые легче интерпретировать машине. Наши данные состоят из различных ключевых слов для действия и изменения состояния, имен объектов, а также произвольных строк текстового чата. Мы хотели бы, чтобы это было разбито на последовательность символов, которые могут быть легко обработанными компилятором. Разбивание действий и текстового чата является более сложным, чем разбивание исходного кода, в связи с тем, что действия относятся к объектам, которые изменяют состояние с течением времени. Ниже мы подробно опишем наш подход к разбиванию физических действий и диалога актов.

Разбивка физических действий

Мы рассматриваем лексику контекстно-зависимых, ролезависиых физических действий по принципу процедуры снизу-вверх. Состояние каждого объекта в игре (например, стейк, кофе, меню, кастрюли, миски и техника) представлена небольшим вектором переменных, в том числе ON, ATTACHED_TO, и SHAPE. (Некоторые объекты меняют свою форму: в результате действий стейк уменьшается с каждым укусом). Наши лексические процедуры анализа отслеживают состояние каждого объекта в течение каждой игровой сессии. Когда лексический анализатор встречает действие (например, PICK_UP), мы предполагаем, что текущее состояние подразумевает изменение состояния, за которым непосредственно следует действие, представляющее эффект. Мы храним каждое уникальное наблюдение действия в лексике. Например, у нас есть одна запись в лексиконе для официантки: взятие пирога с прилавка. Распознавая, что многие объекты выполняют ту же функцию в игре, мы автоматически проверяем объекты по их наблюденным свойствам. Для каждого типа объекта мы рассчитываем, сколько раз объект является целью каждого возможного действия. Из этих счетчиков вычислим вероятность каждого действия с каждым объектом, и проигнорируем действия с вероятностью ниже указанного порога. Объекты, которые в общем списке вероятных действий (допустимости), остаются. Например, мы узнаем, что клиенты, как правило, сидят на стульях и табуретах, и обычно заказывают стейк из лосося, как правило, и официантки чаще всего носят их. Кластеризация объектов значительно снижает размер действий лексики, которая относится к этим объектам. После обработки 5000 логов, наш лексикон содержит 11206 некластеризованных действий, и 7086 кластеризованных. Как только мы узнали лексику, мы можем использовать ее для анализа логов. Каждое действие в журнале может быть заменено индексом действия из лексики, которая служит уникальным ключом. Поведение компилятора может определить задачи, основанные исключительно на маркерах последовательности, без отслежки подробностей об изменении состояния. Разбирая диалоги, непосредственно наблюдаем в

логах - когда официантка берет стейк, идет запись в журнал, стейк теперь ATTACHED_TO к официантке. То же самое нельзя сказать о текстовом чате. Когда клиент говорит "Я съел стейк", нет точного определения этого события в журнале логов. Принимая за основу философские наблюдения, что "что-то говоря, мы что-то делаем "(Остин 1962) мы признаем, что высказывания может служить в качестве речевого акта (Searle 1969), который обслуживает функции, похожие на физические действия, обрабатывая передовое взаимодействие. Однако есть много способов, чтобы сказать ту же самую вещь, используя совершенно другие слова для той же функции. Наш подход к разбору текстового чата опирается на схему классификации, опираясь на то, что кластеры высказывания семантические, так что мы можем представить бесконечное разнообразие способов сказать то же самое с идентичными маркером. Мы классифицируем каждоую линию текстового чата в {речевой акт, содержание, референт}, где речевой акт относится к иллокутивной силе (например, вопрос, директивы, поздравительные открытки), содержание относится к пропозициональному содержанию (в чем вопрос?), и референт идентифицирует объект, на который ссылается высказывание (например, пиво, меню, счет, официантка). Например, "Я хочу пива!" и "Принесите счет" представлены как DIRECTIVE_BRING_BEER. Наша схема имеет 8 возможных речевых актов, 23 содержания, и 16 референтов автора. Члены тройки могут быть помечены OTHER, когда ни одна из существующих этикеток не нужны. В другом месте описанные действия классификатора диалога могут быть обучены автоматически (Оркин & Roy 2010). Для этого исследования, мы вручную аннотируем высказывания о 100 играх, с целью оценки поведения компиляции в изоляции в лучшем сценарии. Мы наблюдаем 312 уникальных диалогов действий в 100 игр. Дополняя действия лексикой с этими диалогами, мы можем представить физические и лингвистические взаимодействий в лог журнала как последовательность лексем, которая может быть обработана равномерно компилятором.

Синтаксический анализ

Имея маркер лога журнала, мы можем перейти к составлению поведения более традиционным способом. Руководствуясь грамматикой спецификации, компилятор выводит структуры, разбирая маркер последовательности. Токены - терминалы грамматики. Синтаксические правила определяют терминалы и нетерминалы (части слева), как любое сочетание терминалов и нетерминалов (части справа), формируя иерархическую структуру. Если маркер не может быть выведенным полностью по грамматике, компилятор вылетает с ошибкой синтаксиса. YACC (Yet Another Compiler) является инструмент Unix, который генерирует парсер данной грамматики (Levin и др. 1992). Порожденный анализатор может включать код пользователя в машинном представлении или промежуточном представлении. YACC грамматика спецификации синтаксиса Java может включают в себя:

```
If_Statement
: IF '(' Expression ')' Statement
| IF '(' Expression ')' Statement ELSE
    Statement
While_Statement
: WHILE '(' Expression ')' Statement
```

Обратите внимание, что здесь может быть несколько действительных последовательностей, представляющих таклй же нетерминала. В грамматике фрагмент выше есть две последовательности предусмотренных If_Statement. Именно такой подход мы принимаем, признавая структуры в поведении - предоставление поведения компилятора с грамматикой для задачи разложения, где каждую задачу можно разложить на любое количество маркеров последовательностей. Например, наша грамматика для ресторана поведение может включать:

```
W_Serves_Food
: W_PICKSUP_FOOD_FROM_COUNTER
    W_PUTSDOWN_FOOD_ON_TABLE
| W_PICKSUP_FOOD_FROM_COUNTER
    W_GIVES_FOOD_TO_C
    C_EXPRESSIVE_THANKS_NONE
| W_PICKSUP_FOOD_FROM_COUNTER
    W_ASSERTION_GIVE_FOOD
   W_PUTSDOWN_FOOD_ON_TABLE
```

В этом примере, маркеры представлены в простом английском языке для читабельности, в то время, как в действительности терминалы грамматики хранятся в виде числовых индексов в лексике. То, что мы описали в данный момент, работает с точно таким же подходом для компиляции исходного кода и поведения. Однако есть некоторые осложнения, связанные с синтаксисом поведения, которые мешают нам непосредственно примененять такой инструмент, как YACC, и требуют альтернативный подход. Первая сложность возникает из-за спонтанного характера логов, которые могут включать синтаксически недействительные последовательности действий, связанные с игроком, нерешительность, или преднамеренное нарушение. Недопустимая последовательность может состоять из действительных маркеров обыкновенных действий и/или высказываний, выполненных в ненормальном контексте, таких, как заказ официантке 50 сортов пива и складывание их на панель вместо обслуживания клиентов. Чтобы не останавливаться на синтаксической ошибке, мы хотели бы обучить компилятор так, чтобы быть в состоянии отделить сигнал от шума, а также пренебречь недопустимой последовательностью, продолжая при этом процесс действительных наблюдений. Вторая трудность связана с тем, что имеет место человеческое взаимодействие игроков. При этом официантка может убирать стол, когда клиент пьет вино. Действия, относящиеся к этим задачам, могут быть сколь угодно взаимосвязанны и могут приводить к несвязанным целям верхнего уровня (например, утоления голода, зарабатывая деньги, словно компилятор сталкивается с кодом из нескольких программ, которые были произвольно объединены в один входной файл, и ему необходимо скомпилировать их все одновременно. Ниже мы описываем наш подход к поведению компиляции при таких осложнениях.